

Engenharia de Software

Uma Abordagem Profissional

Sétima Edição



Roger S. Pressman

**Mc
Graw
Hill**







P934e Pressman, Roger S.
Engenharia de software [recurso eletrônico] : uma abordagem profissional / Roger S. Pressman ; tradução Ariovaldo Griesi ; revisão técnica Reginaldo Arakaki, Julio Arakaki, Renato Manzan de Andrade. – 7. ed. – Dados eletrônicos. – Porto Alegre : AMGH, 2011.

Editado também como livro impresso em 2011.
ISBN 978-85-8055-044-3

1. Engenharia de programas de computador. 2. Gestão de projetos de softwares. I. Título.

CDU 004.41

Engenharia de Software

UMA ABORDAGEM PROFISSIONAL

SÉTIMA EDIÇÃO

Roger S. Pressman, Ph.D.

Tradução

Ariovaldo Griesi

Mario Moro Fecchio

Revisão Técnica

Reginaldo Arakaki

Professor Doutor do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP

Julio Arakaki

Professor Doutor do Departamento de Computação da PUC-SP

Renato Manzan de Andrade

Doutorando do Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP

Versão impressa
desta obra: 2011



AMGH Editora Ltda.

2011

Obra originalmente publicada sob o título
Software Engineering: a Practitioner's Approach, 7th Edition
ISBN 0073375977 / 9780073375977

© 2011, The McGraw-Hill Companies, Inc., New York, NY, EUA

Preparação do original: *Mônica de Aguiar Rocha*
Leitura final: *Vera Lúcia Pereira*
Capa: *Triall Composição Editorial Ltda*, arte sobre capa original
Editora sênior: *Arysinha Jacques Affonso*
Assistente editorial: *César Crivelaro*
Diagramação: *Triall Composição Editorial Ltda*

Reservados todos os direitos de publicação em língua portuguesa à
AMGH Editora Ltda (AMGH Editora é uma parceria entre
Artmed® Editora S.A. e McGraw-Hill Education)
Av. Jerônimo de Ornelas, 670 - Santana
90040-340 Porto Alegre RS
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte,
sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação,
fotocópia, distribuição na Web e outros) sem permissão expressa da Editora.

SÃO PAULO
Av. Embaixador Macedo Soares, 10.735 - Pavilhão 5 - Cond. Espace Center
Vila Anastácio 05095-035 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

O AUTOR

Roger S. Pressman é uma autoridade reconhecida internacionalmente nas tecnologias em melhoria de processos de software e engenharia de software. Por mais de três décadas, trabalhou como engenheiro de software, gerente, professor, autor e consultor, concentrando-se nas questões da engenharia de software. Como profissional técnico e gerente nesta área, trabalhou no desenvolvimento de sistemas CAD/CAM para avançadas aplicações de engenharia e manufatura. Também ocupou cargos com responsabilidade pela programação científica e de sistemas.

Após receber o título de Ph.D. em engenharia da University of Connecticut, Pressman começou a dedicar-se à vida acadêmica ao se tornar professor-adjunto de Engenharia da Computação na University of Bridgeport e diretor do Centro de Projeto e Fabricação Apoiados por Computador (Computer-Aided Design and Manufacturing Center) dessa Universidade.

Atualmente, é presidente da R. S. Pressman & Associates, Inc., uma consultoria especializada em treinamento e métodos em engenharia de software. Atua como consultor-chefe e projetou e desenvolveu o *Essential Software Engineering*, um conjunto de vídeos em engenharia de software, e o *Process Advisor*, um sistema autodirigido para aperfeiçoamento de processos de software. Ambos são usados por milhares de empresas em todo o mundo. Mais recentemente, trabalhou em conjunto com a *EdistaLearning*, na Índia, desenvolvendo extenso treinamento em engenharia de software baseado na Internet.

Publicou vários artigos técnicos, escreve regularmente para periódicos do setor e é autor de sete livros técnicos. Além de *Engenharia de software: uma abordagem profissional*, foi coautor de *Web engineering* (McGraw-Hill), um dos primeiros livros a aplicar um conjunto personalizado de princípios e práticas de engenharia de software para o desenvolvimento de aplicações e sistemas baseados na Web. Também escreveu o premiado *A manager's guide to software engineering* (McGraw-Hill); *Making software engineering happen* (Prentice Hall), primeiro livro a tratar dos críticos problemas de gerenciamento associados ao aperfeiçoamento de processos de software, e *Software shock* (Dorset House), um tratado que se concentra em software e seu impacto sobre os negócios e a sociedade. O dr. Pressman participou dos comitês editoriais de uma série de periódicos do setor e, por muitos anos, foi editor da coluna "Manager" no *IEEE Software*.

É ainda um palestrante renomado, destacando-se em uma série das principais conferências do setor. É associado da IEEE e Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu e Pi Tau Sigma.

Vive no sul da Flórida com sua esposa, Barbara. Atleta por grande parte de sua vida, continua a ser um dedicado jogador de tênis (NTRP 4.5) e jogador de golfe com handicap de apenas um dígito. Nas horas vagas, escreveu dois romances, *The Aymara bridge* e *The Puppeteer*, e planeja começar um novo romance.

*Em memória de meu querido
pai, que viveu 94 anos
e ensinou-me, acima de tudo,
que a honestidade e a integridade
seriam os meus melhores guias na
jornada da vida.*

Quando um software é bem-sucedido — atende às necessidades dos usuários, opera perfeitamente durante um longo período, é fácil de modificar e, mais fácil ainda, de utilizar —, ele é realmente capaz de mudar as coisas para melhor. Porém, quando um software falha — quando seus usuários estão insatisfeitos, quando é propenso a erros, quando é difícil modificá-lo e mais difícil ainda utilizá-lo —, fatos desagradáveis podem e, de fato, acontecem. Todos queremos construir softwares que facilitem o trabalho, evitando pontos negativos latentes nas tentativas mal-sucedidas. Para termos êxito, precisamos de disciplina no projeto e na construção do software. Precisamos de uma abordagem de engenharia.

Já faz quase três décadas que a primeira edição deste livro foi escrita. Durante esse período, a engenharia de software evoluiu de algo obscuro praticado por um número relativamente pequeno de adeptos para uma disciplina de engenharia legítima. Hoje, é reconhecida como uma matéria digna de pesquisa séria, estudo consciencioso e debates acalorados. Neste segmento, o cargo preferido passou a ser o de engenheiro de software e não mais o de programador. Modelos de processos de software, métodos de engenharia de software, bem como ferramentas de software, vêm sendo adotados com sucesso em um amplo espectro de aplicações na indústria.

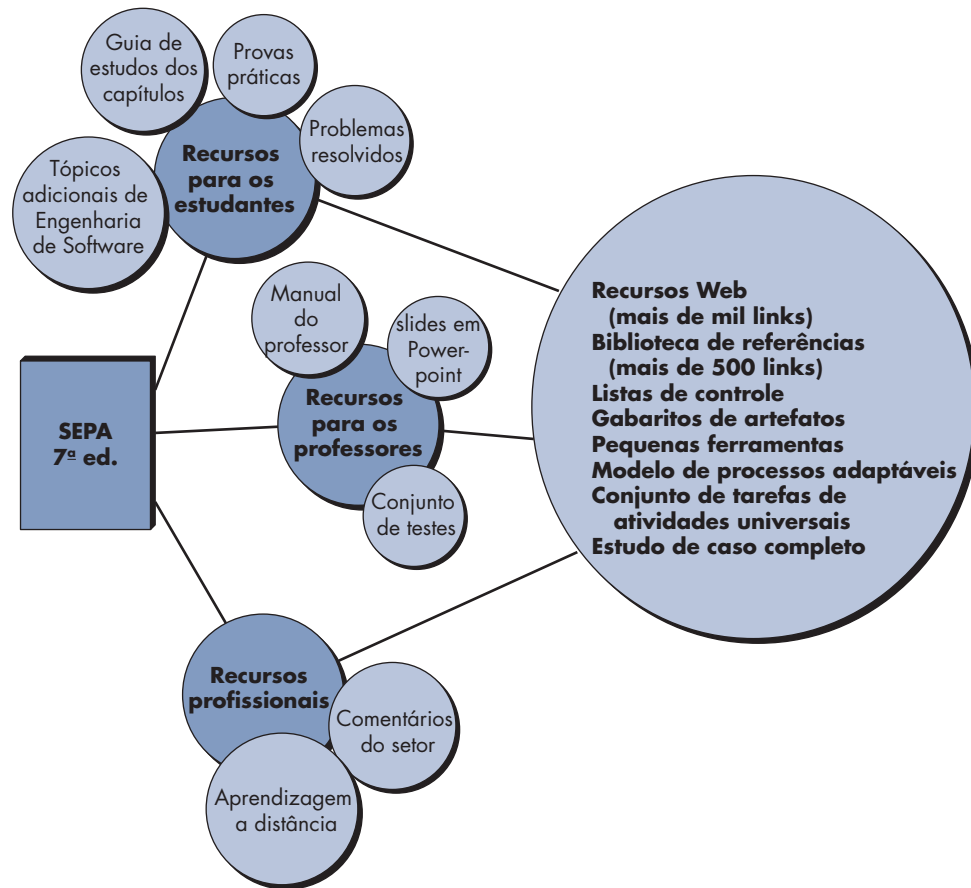
Apesar de gerentes e profissionais envolvidos com a área técnica reconhecerem a necessidade de uma abordagem mais disciplinada em relação ao software, eles continuam a discutir a maneira como essa disciplina deve ser aplicada. Muitos indivíduos e empresas desenvolvem software de forma desordenada, mesmo ao construírem sistemas dirigidos às mais avançadas tecnologias. Grande parte de profissionais e estudantes não estão cientes dos métodos modernos. E, como consequência, a qualidade do software que produzem é afetada. Além disso, a discussão e a controvérsia sobre a real natureza da abordagem de engenharia de software continuam. A engenharia de software é um estudo repleto de contrastes. A postura mudou, progressos foram feitos, porém, falta muito para essa disciplina atingir a maturidade.

A sétima edição do livro *Engenharia de software: uma abordagem profissional* é destinada a servir de guia para uma disciplina de engenharia em maturação. Assim como nas seis edições anteriores, esta é voltada tanto para estudantes no final do curso de graduação ou no primeiro ano de pós-graduação e para profissionais da área.

A sétima edição é muito mais do que uma simples atualização. O livro foi revisado e reestruturado para adquirir maior fluência em termos pedagógicos e enfatizar novos e importantes processos e práticas de engenharia de software. Além disso, um “sistema de suporte” revisado e atualizado, ilustrado na figura da página seguinte, fornece um conjunto completo de recursos para estudantes, professores e profissionais complementarem o conteúdo do livro. Tais recursos são apresentados como parte do site do livro em inglês (www.mhhe.com/pressman), especificamente projetado para este livro.

A Sétima Edição. Os 32 capítulos foram reorganizados em cinco partes, que diferem consideravelmente da sexta edição para melhor compartimentalizar tópicos e auxiliar os professores que talvez não tenham tempo suficiente para abordar o livro inteiro em um semestre.

**Sistema de
suporte para a
7ª ed.**



A Parte 1, *O Processo*, apresenta uma série de visões diferentes de processo de software, considerando todos os modelos importantes e contemplando o debate entre as filosofias de processos ágeis e preceptivos. A Parte 2, *Modelagem*, fornece métodos de projeto e análise com ênfase em técnicas orientadas a objetos e modelagem UML. O projeto baseado em padrões, bem como o projeto para aplicações da Web, também é considerado. A Parte 3, *Gestão da Qualidade*, apresenta conceitos, procedimentos, técnicas e métodos que permitem a uma equipe de software avaliar a qualidade de software, revisar produtos gerados por engenharia de software, realizar procedimentos SQA e aplicar uma estratégia e tática de testes eficazes. Além disso, são considerados também métodos de modelagem e verificação formais. A Parte 4, *Gerenciamento de Projetos de Software*, aborda tópicos relevantes para os que planejam, gerenciam e controlam um projeto de desenvolvimento de software. A Parte 5, *Tópicos Avançados*, considera aperfeiçoamento de processos de software e tendências da engenharia de software. Preservando a tradição de edições anteriores, usa-se uma série de quadros ao longo do livro para apresentar as atribuições de uma equipe de desenvolvimento (fictícia), bem como fornecer material suplementar sobre métodos e ferramentas relevantes para os tópicos do capítulo. Dois novos apêndices fornecem tutoriais curtos sobre a filosofia de orientação a objetos e UML para os leitores que não estão familiarizados com esses itens.

A organização da edição em cinco partes possibilita ao professor “aglutinar” tópicos tomando como base o tempo disponível e a necessidade dos alunos. Por exemplo, cursos de um semestre podem ser montados baseando-se em uma ou mais das cinco partes; cursos que ofereçam uma visão geral da engenharia de software selecionariam capítulos de todas as cinco

partes; outro, sobre engenharia de software que enfatize análise e projeto, utilizaria tópicos das Partes 1 e 2; cursos de engenharia de software voltado para testes enfatizaria tópicos das Partes 1 e 3, com uma breve incursão na Parte 2; já “cursos voltados para administradores” concentrariam-se nas Partes 1 e 4. Organizada dessa maneira, a sétima edição oferece ao professor uma série de opções didáticas. Esse conteúdo é complementado pelos seguintes elementos.

Recursos para os estudantes. A ampla gama de instrumentos para os estudantes inclui um completo centro de aprendizagem on-line englobando guias de estudo capítulo por capítulo, testes práticos, soluções de problemas e uma série de recursos baseados na Web, incluindo listas de controle de engenharia de software, um conjunto em constante evolução de “pequenas ferramentas”, um estudo de caso completo, gabaritos de artefatos e muitos outros. Há, ainda, mais de 1.000 *Referências na Web*, classificadas por categoria, que permitem ao estudante explorar a engenharia de software de forma mais detalhada, e uma *Biblioteca de Referências* com links de mais de 500 artigos, os quais fornecem uma extensa fonte de informações avançadas sobre engenharia de software.

Recursos para os professores. Desenvolveu-se uma ampla gama de recursos para os professores para suplementar a sétima edição. Entre eles: um completo *Guia do Instrutor on-line* (que também pode ser baixado) e materiais didáticos, como, por exemplo, um conjunto completo com mais de 700 *slides em PowerPoint* que pode ser usado em aulas, além de uma série de testes. Também estão disponíveis todos os recursos dirigidos aos estudantes (como as pequenas ferramentas, as Referências na Web, a Biblioteca de Referências que pode ser baixada) e os recursos para profissionais.

O *Guia do Instrutor* apresenta sugestões para a realização de vários tipos de cursos de engenharia de software, recomendações para uma variedade de projetos de software a ser realizados com um curso, soluções para problemas escolhidos e uma série de ferramentas didáticas úteis.

Recursos para os profissionais. No conjunto de recursos disponíveis para profissionais da área (bem como para estudantes e corpo docente), temos resumos e exemplos de documentos de engenharia de software e outros artefatos, um útil conjunto de listas de controle de engenharia de software, um catálogo de ferramentas de engenharia de software (CASE), uma completa coleção de recursos baseados na Web e um “modelo de processos adaptáveis”, que compõe de forma detalhada as tarefas do processo de engenharia de software.

Quando associada ao seu sistema de suporte on-line, esta edição gera flexibilidade e profundidade do conteúdo que não poderiam ser atingidas por um livro-texto isolado.

Agradecimentos. Meu trabalho nas sete edições de *Engenharia de software: uma abordagem profissional* tem sido o mais longo e contínuo projeto técnico de minha vida. Mesmo quando termino a atividade de redação, informações extraídas da literatura técnica continuam a ser assimiladas e organizadas, bem como sugestões e críticas de leitores ao redor do mundo são avaliadas e catalogadas. Por essa razão, meus agradecimentos aos muitos autores de livros e artigos acadêmicos (tanto em papel quanto em meio eletrônico) que me forneceram visões, ideias e comentários adicionais ao longo de aproximadamente 30 anos.

Agradecimentos especiais a Tim Lethbridge, da University of Ottawa, que me ajudou no desenvolvimento de exemplos em UML e OCL e a desenvolver o estudo de caso que acompanha este livro, bem como Dale Skrien do Colby College, que desenvolveu o tutorial UML do Apêndice 1. Sua ajuda e comentários foram inestimáveis. Um agradecimento especial a Bruce Maxim, da University of Michigan–Dearborn, que me ajudou a desenvolver grande parte do conteúdo do site pedagógico que acompanha a edição em inglês. Por fim, gostaria de agradecer aos revisores da sétima edição: seus comentários profundos e críticas inteligentes foram valiosas.

Osman Balci,
Virginia Tech University

Max Fomitchev,
Penn State University

Jerry (Zeyu) Gao,
San Jose State University

Guillermo Garcia,
Universidad Alfonso X Madrid

Pablo Gervas,
Universidad Complutense de Madrid

SK Jain,
National Institute of Technology Hamirpur

Saeed Monemi,
Cal Poly Pomona

Ahmed Salem,
California State University

Vasudeva Varma,
IIIT Hyderabad

O conteúdo da sétima edição de *Engenharia de software: uma abordagem profissional* foi moldado por profissionais da área, professores universitários e estudantes que usaram edições anteriores do livro e se deram ao trabalho de enviar sugestões, críticas e ideias. Meus agradecimentos a cada um de vocês. Além disso, meus agradecimentos pessoais vão a muitas pessoas da indústria, distribuídas por todo o mundo, que me ensinaram tanto ou mais do que eu poderia ensinar-lhes.

Assim como as edições deste livro evoluíram, meus filhos, Mathew e Michael, cresceram e se tornaram homens. Sua maturidade, caráter e sucesso me inspiraram. Nada mais me deixou tão orgulhoso. E, finalmente, para você, Barbara, meu amor, agradeço por ter tolerado as várias horas que dediquei ao trabalho e por ter me estimulado para mais uma edição do “livro”.

Roger S. Pressman

SUMÁRIO RESUMIDO

CAPÍTULO 1 Engenharia de Software 29

PARTE UM PROCESSOS DE SOFTWARE 51

CAPÍTULO 2 Modelos de Processo 52

CAPÍTULO 3 Desenvolvimento Ágil 81

PARTE DOIS MODELAGEM 107

CAPÍTULO 4 Princípios que Orientam a Prática 108

CAPÍTULO 5 Engenharia de Requisitos 126

CAPÍTULO 6 Modelagem de Requisitos: Cenários, Informações e Classes de Análise 150

CAPÍTULO 7 Modelagem de Requisitos: Fluxo, Comportamento, Padrões e Aplicações Baseadas na Web (WebApp) 181

CAPÍTULO 8 Conceitos de Projeto 206

CAPÍTULO 9 Projeto de Arquitetura 229

CAPÍTULO 10 Projeto de Componentes 257

CAPÍTULO 11 Projeto de Interfaces do Usuário 287

CAPÍTULO 12 Projeto Baseado em Padrões 316

CAPÍTULO 13 Projeto de WebApps 338

PARTE TRÊS GESTÃO DA QUALIDADE 357

CAPÍTULO 14 Conceitos de Qualidade 358

CAPÍTULO 15 Técnicas de Revisão 373

CAPÍTULO 16 Garantia da Qualidade de Software 387

CAPÍTULO 17 Estratégias de Teste de Software 401

CAPÍTULO 18 Testando Aplicativos Convencionais 428

CAPÍTULO 19 Testando Aplicações Orientadas a Objeto 453

CAPÍTULO 20 Testando Aplicações para Web 468

CAPÍTULO 21 Modelagem Formal e Verificação 491

CAPÍTULO 22 Gestão de Configuração de Software 514

CAPÍTULO 23 Métricas de Produto 538

PARTE QUATRO GERENCIAMENTO DE PROJETOS DE SOFTWARE 565

CAPÍTULO 24 Conceitos de Gerenciamento de Projeto 566

CAPÍTULO 25 Métricas de Processo e Projeto 583

CAPÍTULO 26 Estimativas de Projeto de Software 604

CAPÍTULO 27 Cronograma de Projeto 629

CAPÍTULO 28	Gestão de Risco	648
CAPÍTULO 29	Manutenção e Reengenharia	662

PARTE CINCO	TÓPICOS AVANÇADOS	681
--------------------	--------------------------	-----

CAPÍTULO 30	Melhoria do Processo de Software	682
CAPÍTULO 31	Tendências Emergentes na Engenharia de Software	701
CAPÍTULO 32	Comentários Finais	721
APÊNDICE 1	Introdução à UML	727
APÊNDICE 2	Conceitos Orientados a Objeto	744
REFERÊNCIAS		751
ÍNDICE		773

CAPÍTULO 1 ENGENHARIA DE SOFTWARE 29

- 1.1 A Natureza do Software 31
 - 1.1.1 Definindo software 32
 - 1.1.2 Campos de aplicação de software 34
 - 1.1.3 Software legado 36
- 1.2 A Natureza Única das Webapps 37
- 1.3 Engenharia de Software 38
- 1.4 O Processo de Software 40
- 1.5 A Prática da Engenharia de Software 42
 - 1.5.1 A essência da prática 42
 - 1.5.2 Princípios gerais 44
- 1.6 Mitos Relativos ao Software 45
- 1.7 Como Tudo Começou 47
- 1.8 Resumo 48
- Problemas e Pontos a Ponderar 49
- Leituras e Fontes de Informação Complementares 49

PARTE UM PROCESSOS DE SOFTWARE 51

CAPÍTULO 2 MODELOS DE PROCESSO 52

- 2.1 Um Modelo de Processo Genérico 53
 - 2.1.1 Definindo atividade metodológica 55
 - 2.1.2 Identificação de um conjunto de tarefas 55
 - 2.1.3 Padrões de processos 55
- 2.2 Avaliação e Aperfeiçoamento de Processos 58
- 2.3 Modelos de Processo Prescritivo 58
 - 2.3.1 O modelo cascata 59
 - 2.3.2 Modelos de processo incremental 61
 - 2.3.3 Modelos de processo evolucionário 62
 - 2.3.4 Modelos concorrentes 67
 - 2.3.5 Um comentário final sobre processos evolucionários 68
- 2.4 Modelos de Processo Especializado 69
 - 2.4.1 Desenvolvimento baseado em componentes 69
 - 2.4.2 O modelo de métodos formais 69
 - 2.4.3 Desenvolvimento de software orientado a aspectos 70
- 2.5 O Processo Unificado 71
 - 2.5.1 Breve histórico 72
 - 2.5.2 Fases do processo unificado 72
- 2.6 Modelos de Processo Pessoal e de Equipe 74
 - 2.6.1 Processo de Software Pessoal (PSP) 74
 - 2.6.2 Processo de Software em Equipe (TSP) 75
- 2.7 Tecnologia de Processos 76
- 2.8 Processo do Produto 77
- 2.9 Resumo 78
- Problemas e Pontos a Ponderar 78
- Leituras e Fontes de Informação Complementares 79

CAPÍTULO 3 DESENVOLVIMENTO ÁGIL 81

- 3.1 O que é Agilidade? 82
- 3.2 Agilidade e o Custo das Mudanças 83
- 3.3 O que é Processo Ágil? 84
 - 3.3.1 Princípios da agilidade 84
 - 3.3.2 A política do desenvolvimento ágil 85
 - 3.3.3 Fatores humanos 86
- 3.4 Extreme Programming – XP (Programação Extrema) 87
 - 3.4.1 Valores da XP 87
 - 3.4.2 O processo XP 88
 - 3.4.3 Industrial XP 91
 - 3.4.4 O debate XP 92
- 3.5 Outros Modelos de Processos Ágeis 93
 - 3.5.1 Desenvolvimento de Software Adaptativo (ASD) 94
 - 3.5.2 Scrum 95
 - 3.5.3 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM) 96
 - 3.5.4 Crystal 97
 - 3.5.5 Desenvolvimento Dirigido a Funcionalidades (FDD) 98
 - 3.5.6 Desenvolvimento de Software Enxuto (LSD) 99
 - 3.5.7 Modelagem Ágil (AM) 99
 - 3.5.8 Processo Unificado Ágil (AUP) 101
- 3.6 Um Conjunto de Ferramentas para o Processo Ágil 102
- 3.7 Resumo 102
- Problemas e Pontos a Ponderar 103
- Leituras e Fontes de Informação Complementares 104

PARTE DOIS**MODELAGEM 107**

CAPÍTULO 4 PRINCÍPIOS QUE ORIENTAM A PRÁTICA 108

- 4.1 Conhecimento da Engenharia de Software 109
- 4.2 Princípios Fundamentais 109
 - 4.2.1 Princípios que orientam o processo 110
 - 4.2.2 Princípios que orientam a prática 111
- 4.3 Princípios das Atividades Metodológicas 112
 - 4.3.1 Princípios da comunicação 112
 - 4.3.2 Princípios de planejamento 114
 - 4.3.3 Princípios de modelagem 116
 - 4.3.4 Princípios de construção 120
 - 4.3.5 Princípios de disponibilização 122
- 4.4 Resumo 123
- Problemas e Pontos a Ponderar 123
- Leituras e Fontes de Informação Complementares 124

CAPÍTULO 5 ENGENHARIA DE REQUISITOS 126

- 5.1 Engenharia de Requisitos 127
- 5.2 Início do Processo de Engenharia de Requisitos 131
 - 5.2.1 Identificação de interessados 131
 - 5.2.2 Reconhecimento de diversos pontos de vista 131
 - 5.2.3 Trabalho na busca da colaboração 132
 - 5.2.4 Perguntas iniciais 132

5.3	Levantamento de Requisitos	133
5.3.1	Coleta colaborativa de requisitos	133
5.3.2	Disponibilização da função de qualidade	136
5.3.3	Cenários de uso	136
5.3.4	Artefatos do levantamento de requisitos	137
5.4	Desenvolvimento de Casos de Uso	137
5.5	Construção do modelo de análise	142
5.5.1	Elementos do modelo de análise	142
5.5.2	Padrões de análise	145
5.6	Negociação de Requisitos	145
5.7	Validação dos Requisitos	146
5.8	Resumo	147
	Problemas e Pontos a Ponderar	147
	Leituras e Fontes de Informação Complementares	148

CAPÍTULO 6 MODELAGEM DE REQUISITOS: CENÁRIOS, INFORMAÇÕES E CLASSES DE ANÁLISE 150

6.1	Análise de Requisitos	151
6.1.1	Filosofia e objetivos gerais	152
6.1.2	Regras práticas para a análise	152
6.1.3	Análise de domínio	153
6.1.4	Abordagens à modelagem de requisitos	154
6.2	Modelagem Baseada em Cenários	155
6.2.1	Criação de um caso de uso preliminar	155
6.2.2	Refinamento de um caso de uso preliminar	158
6.2.3	Criação de um caso de uso formal	159
6.3	Modelos UML que Complementam o Caso de Uso	161
6.3.1	Desenvolvimento de um diagrama de atividade	161
6.3.2	Diagramas de raias	162
6.4	Conceitos de Modelagem de Dados	163
6.4.1	Objetos de dados	163
6.4.2	Atributos de dados	163
6.4.3	Relacionamentos	164
6.5	Modelagem Baseada em Classes	166
6.5.1	Identificação de classes de análise	166
6.5.2	Especificação de atributos	169
6.5.3	Definição das operações	170
6.5.4	Modelagem CRC (Classe-Responsabilidade-Colaborador)	171
6.5.5	Associações e dependências	176
6.5.6	Pacotes de análise	178
6.6	Resumo	178
	Problemas e Pontos a Ponderar	179
	Leituras e Fontes de Informação Complementares	180

CAPÍTULO 7 MODELAGEM DE REQUISITOS: FLUXO, COMPORTAMENTO, PADRÕES E APLICAÇÕES BASEADAS NA WEB (WEBAPP) 181

7.1	Estratégias de Modelagem de Requisitos	182
7.2	Modelagem Orientada a Fluxos	182
7.2.1	Criação de um modelo de fluxo de dados	182
7.2.2	Criação de um modelo de fluxo de controle	185
7.2.3	A especificação de controles	185
7.2.4	A especificação de processos	186

7.3	Criação de um Modelo Comportamental	188
7.3.1	Identificação de eventos com o caso de uso	189
7.3.2	Representações de estados	189
7.4	Padrões Para a Modelagem de Requisitos	192
7.4.1	Descoberta de padrões de análise	192
7.4.2	Exemplo de padrão de requisitos: atuador-sensor	193
7.5	Modelagem de Requisitos para WebApps	197
7.5.1	Que nível de análise é suficiente?	197
7.5.2	Entrada da modelagem de requisitos	197
7.5.3	Saída da modelagem de requisitos	198
7.5.4	Modelo de conteúdo para WebApps	199
7.5.5	Modelo de interações para WebApps	200
7.5.6	Modelo funcional para WebApps	200
7.5.7	Modelos de configuração para WebApps	201
7.5.8	Modelo de navegação	202
7.6	Resumo	203
	Problemas e Pontos a Ponderar	204
	Leituras e Fontes de Informação Complementares	204

CAPÍTULO 8 CONCEITOS DE PROJETO 206

8.1	Projeto no Contexto da Engenharia de Software	207
8.2	O Processo de Projeto	209
8.2.1	Diretrizes e atributos da qualidade de software	209
8.2.2	A evolução de um projeto de software	211
8.3	Conceitos de Projeto	212
8.3.1	Abstração	212
8.3.2	Arquitetura	213
8.3.3	Padrões	214
8.3.4	Separação por interesses (por afinidades)	214
8.3.5	Modularidade	214
8.3.6	Encapsulamento de informações	215
8.3.7	Independência funcional	216
8.3.8	Refinamento	217
8.3.9	Aspectos	217
8.3.10	Refatoração	218
8.3.11	Conceitos de projeto orientado a objetos	218
8.3.12	Classes de projeto	218
8.4	O Modelo de Projeto	221
8.4.1	Elementos de projeto de dados	222
8.4.2	Elementos de projeto de arquitetura	222
8.4.3	Elementos de projeto de interfaces	222
8.4.4	Elementos de projeto de componentes	224
8.4.5	Elementos de projeto de implantação	224
8.5	Resumo	226
	Problemas e Pontos a Ponderar	226
	Leituras e Fontes de Informação Complementares	227

CAPÍTULO 9 PROJETO DA ARQUITETURA 229

9.1	Arquitetura de Software	230
9.1.1	O que é arquitetura?	230
9.1.2	Por que a arquitetura é importante?	231

9.1.3	Descrições de arquitetura	231
9.1.4	Decisões de arquitetura	232
9.2	Gêneros de Arquitetura	232
9.3	Estilos de Arquitetura	234
9.3.1	Uma breve taxonomia dos estilos de arquitetura	235
9.3.2	Padrões de arquitetura	238
9.3.3	Organização e refinamento	239
9.4	Projeto da Arquitetura	239
9.4.1	Representação do sistema no contexto	240
9.4.2	Definição de arquétipos	241
9.4.3	Refinamento da arquitetura em componentes	242
9.4.4	Descrição das instâncias	243
9.5	Avaliação de Projetos de Arquiteturas Alternativas	244
9.5.1	Um método de análise dos prós e contras de uma arquitetura	245
9.5.2	Complexidade da arquitetura	246
9.5.3	Linguagens de descrição da arquitetura	247
9.6	Mapeamento de Arquitetura Utilizando Fluxo de Dados	247
9.6.1	Mapeamento de transformação	248
9.6.2	Refinamento do projeto da arquitetura	254
9.7	Resumo	255
	Problemas e Pontos a Ponderar	255
	Leituras e Fontes de Informação Complementares	256

CAPÍTULO 10 PROJETO DE COMPONENTES 257

10.1	O Que é Componente?	258
10.1.1	Uma visão orientada a objetos	258
10.1.2	A visão tradicional	260
10.1.3	Uma visão relacionada com processos	262
10.2	Projeto de Componentes Baseados em Classes	262
10.2.1	Princípios básicos de projeto	262
10.2.2	Diretrizes para o projeto de componentes	265
10.2.3	Coesão	266
10.2.4	Acoplamento	267
10.3	Condução de Projetos de Componentes	269
10.4	Projeto de Componentes para WebApps	274
10.4.1	Projeto de conteúdo para componentes	274
10.4.2	Projeto funcional para componentes	275
10.5	Projeto de Componentes Tradicionais	275
10.5.1	Notação gráfica em projeto	276
10.5.2	Notação tabular de projeto	277
10.5.3	Linguagem de projeto de programas	278
10.6	Desenvolvimento Baseado em Componentes	279
10.6.1	Engenharia de domínio	280
10.6.2	Qualificação, adaptação e composição de componentes	280
10.6.3	Análise e projeto para reutilização	282
10.6.4	Classificação e recuperação de componentes	283
10.7	Resumo	284
	Problemas e Pontos a Ponderar	285
	Leituras e Fontes de Informação Complementares	286

CAPÍTULO 11 PROJETO DE INTERFACES DO USUÁRIO 287

- 11.1 As Regras de Ouro 288
 - 11.1.1 Deixar o usuário no comando 288
 - 11.1.2 Reduzir a carga de memória do usuário 289
 - 11.1.3 Tornar a interface consistente 290
- 11.2 Análise e Projeto de Interfaces 291
 - 11.2.1 Modelos de análise e projeto de interfaces 291
 - 11.2.2 O processo 292
- 11.3 Análise de Interfaces 294
 - 11.3.1 Análise de usuários 294
 - 11.3.2 Análise e modelagem de tarefas 295
 - 11.3.3 Análise do conteúdo exibido 299
 - 11.3.4 Análise do ambiente de trabalho 300
- 11.4 Etapas no Projeto de Interfaces 300
 - 11.4.1 Aplicação das etapas para projeto de interfaces 301
 - 11.4.2 Padrões de projeto de interfaces do usuário 302
 - 11.4.3 Questões de projeto 303
- 11.5 Projeto de Interfaces para WebApp 306
 - 11.5.1 Princípios e diretrizes para projeto de interfaces 306
 - 11.5.2 Fluxo de trabalho de projeto de interfaces para WebApps 310
- 11.6 Avaliação de Projeto 311
- 11.7 Resumo 313
- Problemas e Pontos a Ponderar 313
- Leituras e Fontes de Informação Complementares 314

CAPÍTULO 12 PROJETO BASEADO EM PADRÕES 316

- 12.1 Padrão de Projeto 317
 - 12.1.1 Tipos de padrões 318
 - 12.1.2 Estruturas de uso de padrões de projeto 320
 - 12.1.3 Descrição de padrões 320
 - 12.1.4 Linguagens e repositórios de padrões 321
- 12.2 Projeto de Software Baseado em Padrões 322
 - 12.2.1 Contexto do projeto baseado em padrões 322
 - 12.2.2 Pensando em termos de padrões 323
 - 12.2.3 Tarefas de projeto 324
 - 12.2.4 Construção de uma tabela para organização de padrões 325
 - 12.2.5 Erros comuns de projeto 326
- 12.3 Padrões de Arquitetura 327
- 12.4 Padrões de Projeto de Componentes 329
- 12.5 Padrões de Projeto para Interfaces do Usuário 331
- 12.6 Padrões de Projeto para WebApps 333
 - 12.6.1 Foco do projeto 333
 - 12.6.2 Granularidade do projeto 334
- 12.7 Resumo 335
- Problemas e Pontos a Ponderar 336
- Leituras e Fontes de Informação Complementares 336

CAPÍTULO 13 PROJETO DE WEBAPPS 338

- 13.1 Qualidade de Projeto em WebApps 339
- 13.2 Objetivos de Projeto 341
- 13.3 Uma Pirâmide de Projeto para WebApps 342

13.4	Projeto de Interfaces para WebApps	342
13.5	Projeto Estético	344
13.5.1	Problemas de layout	344
13.5.2	Questões de design gráfico	344
13.6	Projeto de Conteúdo	345
13.6.1	Objetos de conteúdo	345
13.6.2	Questões de projeto de conteúdo	345
13.7	Projeto Arquitetural	346
13.7.1	Arquitetura de conteúdo	347
13.7.2	Arquitetura de uma WebApp	349
13.8	Projeto da Navegação	350
13.8.1	Semântica de navegação	350
13.8.2	Sintaxe de navegação	351
13.9	Projeto dos Componentes	352
13.10	Método de Projeto de Hipermídia Orientado a Objetos (<i>Object-Oriented Hypermedia Design Method</i> – OOHDM)	352
13.10.1	Projeto conceitual para o OOHDM	352
13.10.2	Projeto da navegação para o OOHDM	353
13.10.3	Projeto da interface abstrata e implementação	353
13.11	Resumo	354
	Problemas e Pontos a Ponderar	355
	Leituras e Fontes de Informação Complementares	356

PARTE TRÊS**GESTÃO DE QUALIDADE 357**

CAPÍTULO 14 CONCEITOS DE QUALIDADE 358

14.1	O Que é Qualidade?	359
14.2	Qualidade de Software	360
14.2.1	Dimensões de qualidade de Garvin	360
14.2.2	Fatores de qualidade de McCall	361
14.2.3	Fatores de qualidade ISO 9126	362
14.2.4	Fatores de qualidade desejados	363
14.2.5	A transição para uma visão quantitativa	364
14.3	O Dilema da Qualidade de Software	365
14.3.1	Software “bom o suficiente”	365
14.3.2	Custo da qualidade	366
14.3.3	Riscos	368
14.3.4	Negligência e responsabilidade civil	368
14.3.5	Qualidade e segurança	368
14.3.6	O impacto das ações administrativas	369
14.4	Alcançando a Qualidade de Software	370
14.4.1	Métodos de engenharia de software	370
14.4.2	Técnicas de gerenciamento de software	370
14.4.3	Controle de qualidade	370
14.4.4	Garantia da qualidade	370
14.5	Resumo	371
	Problemas e Pontos a Ponderar	371
	Leituras e Fontes de Informação Complementares	372

CAPÍTULO 15 TÉCNICAS DE REVISÃO 373

15.1	Impacto de Defeitos de Software nos Custos	374
15.2	Amplificação e Eliminação de Defeitos	375

15.3	Métricas de Revisão e seu Emprego	376
15.3.1	Análise de métricas	377
15.3.2	Eficácia dos custos de revisões	377
15.4	Revisões: Um Espectro de Formalidade	379
15.5	Revisões Informais	380
15.6	Revisões Técnicas Formais	381
15.6.1	Uma reunião de revisão	381
15.6.2	Relatório de revisão e manutenção de registros	382
15.6.3	Diretrizes de revisão	382
15.6.4	Revisões por amostragem	384
15.7	Resumo	385
	Problemas e Pontos a Ponderar	385
	Leituras e Fontes de Informação Complementares	386

CAPÍTULO 16 GARANTIA DA QUALIDADE DE SOFTWARE 387

16.1	Problemas de Background	388
16.2	Elementos de Garantia da Qualidade de Software	388
16.3	Tarefas, Metas e Métricas da SQA	390
16.3.1	Tarefas da SQA	390
16.3.2	Metas, atributos e métricas	391
16.4	Abordagens Formais da SQA	392
16.5	Estatística da Garantia da Qualidade de Software	393
16.5.1	Um exemplo genérico	393
16.5.2	Seis sigma para engenharia de software	394
16.6	Confiabilidade de Software	395
16.6.1	Medidas de confiabilidade e disponibilidade	395
16.6.2	Proteção do software	396
16.7	Os Padrões de Qualidade Iso 9000	397
16.8	O Plano de Sqa	398
16.9	Resumo	399
	Problemas e Pontos a Ponderar	399
	Leituras e Fontes de Informação Complementares	400

CAPÍTULO 17 ESTRATÉGIAS DE TESTE DE SOFTWARE 401

17.1	Uma Abordagem Estratégica do Teste de Software	402
17.1.1	Verificação e validação	402
17.1.2	Organizando o teste de software	403
17.1.3	Estratégia de teste de software — a visão ampla	404
17.1.4	Critérios para conclusão do teste	405
17.2	Problemas Estratégicos	406
17.3	Estratégias Para Software Convencional	407
17.3.1	Teste de unidade	407
17.3.2	Teste de integração	409
17.4	Estratégias de Teste Para Software Orientado a Objeto	415
17.4.1	Teste de unidade no contexto OO	415
17.4.2	Teste de integração no contexto OO	415
17.5	Estratégias de Teste Para WebApps	416
17.6	Teste de Validação	416
17.6.1	Critério de teste de validação	417
17.6.2	Revisão da configuração	417
17.6.3	Teste alfa e beta	417

17.7	Teste de Sistema	418
17.7.1	Teste de recuperação	419
17.7.2	Teste de segurança	419
17.7.3	Teste por esforço	419
17.7.4	Teste de desempenho	420
17.7.5	Teste de disponibilização	420
17.8	A Arte da Depuração	421
17.8.1	O processo de depuração	421
17.8.2	Considerações psicológicas	422
17.8.3	Estratégias de depuração	423
17.8.4	Correção do erro	424
17.9	Resumo	425
	Problemas e Pontos a Ponderar	425
	Leituras e Fontes de Informação Complementares	426

CAPÍTULO 18 TESTANDO APLICATIVOS CONVENCIONAIS 428

18.1	Fundamentos do Teste de Software	429
18.2	Visões Interna e Externa do Teste	430
18.3	Teste Caixa Branca	431
18.4	Teste do Caminho Básico	431
18.4.1	Notação de grafo de fluxo	432
18.4.2	Caminhos de programa independentes	433
18.4.3	Derivação de casos de teste	435
18.4.4	Matrizes de grafos	436
18.5	Teste de Estrutura de Controle	437
18.5.1	Teste de condição	437
18.5.2	Teste de fluxo de dados	438
18.5.3	Teste de ciclo	438
18.6	Teste Caixa Preta	439
18.6.1	Métodos de teste com base em grafo	440
18.6.2	Particionamento de equivalência	441
18.6.3	Análise de valor limite	442
18.6.4	Teste de matriz ortogonal	442
18.7	Teste Baseado em Modelos	445
18.8	Teste Para Ambientes, Arquiteturas e Aplicações Especializados	446
18.8.1	Testando GUIs	446
18.8.2	Teste de arquiteturas cliente-servidor	446
18.8.3	Testando a documentação e os recursos de ajuda	447
18.8.4	Teste para sistema em tempo real	448
18.9	Padrões para Teste de Software	449
18.10	Resumo	450
	Problemas e Pontos a Ponderar	451
	Leituras e Fontes de Informação Complementares	452

CAPÍTULO 19 TESTANDO APLICAÇÕES ORIENTADAS A OBJETO 453

19.1	Ampliando a Versão do Teste	454
19.2	Testando Modelos de Análise Orientada a Objeto (OOA) e Projeto Orientado a Objeto (OOD)	455
19.2.1	Exatidão dos modelos de OOA e OODs	455
19.2.2	Consistência dos modelos orientados a objeto	455
19.3	Estratégias de Teste Orientado a Objeto	457
19.3.1	Teste de unidade em contexto orientado a objeto	457

19.3.2	Teste de integração em contexto orientado a objeto	457
19.3.3	Teste de validação em contexto orientado a objeto	458
19.4	Métodos de Teste Orientados a Objeto	458
19.4.1	As implicações no projeto de casos de teste dos conceitos orientados a objeto	459
19.4.2	Aplicabilidade dos métodos convencionais de projeto de casos de teste	459
19.4.3	Teste baseado em falhas	459
19.4.4	Casos de teste e a hierarquia de classe	460
19.4.5	Projeto de teste baseado em cenário	460
19.4.6	Teste da estrutura superficial e estrutura profunda	462
19.5	Métodos de Teste Aplicáveis no Nível de Classe	462
19.5.1	Teste aleatório para classes orientadas a objeto	462
19.5.2	Teste de partição em nível de classe	463
19.6	Projeto de Caso de Teste Interclasse	464
19.6.1	Teste de múltiplas classes	464
19.6.2	Testes derivados de modelos comportamentais	465
19.7	Resumo	466
	Problemas e Pontos a Ponderar	467
	Leituras e Fontes de Informação Complementares	467

CAPÍTULO 20 TESTANDO APLICAÇÕES PARA WEB 468

20.1	Conceitos de Teste para WebApps	469
20.1.1	Dimensões da qualidade	469
20.1.2	Erros em um ambiente WebApp	469
20.1.3	Estratégia de teste	470
20.1.4	Planejamento de teste	470
20.2	O Processo de Teste – uma Visão Geral	471
20.3	Teste de Conteúdo	472
20.3.1	Objetivos do teste de conteúdo	472
20.3.2	Teste de base de dados	473
20.4	Teste da Interface de Usuário	474
20.4.1	Estratégia de teste de interface	474
20.4.2	Testando mecanismos de interface	475
20.4.3	Testando semânticas de interface	477
20.4.4	Testes de usabilidade	477
20.4.5	Testes de compatibilidade	478
20.5	Teste no Nível de Componente	479
20.6	Testes de Navegação	480
20.6.1	Testando a sintaxe de navegação	481
20.6.2	Testando as semânticas de navegação	481
20.7	Teste de Configuração	482
20.7.1	Tópicos no lado do servidor	483
20.7.2	Tópicos no lado do cliente	483
20.8	Teste de Segurança	484
20.9	Teste de Desempenho	485
20.9.1	Objetivos do teste de desempenho	485
20.9.2	Teste de carga	486
20.9.3	Teste de esforço (<i>stress</i>)	486
20.10	Resumo	487
	Problemas e Pontos a Ponderar	488
	Leituras e Fontes de Informação Complementares	489

CAPÍTULO 21 MODELAGEM FORMAL E VERIFICAÇÃO 491

- 21.1 Estratégia Sala Limpa 492
- 21.2 Especificação Funcional 493
 - 21.2.1 Especificação caixa preta 495
 - 21.2.2 Especificação caixa de estado 495
 - 21.2.3 Especificação caixa clara 495
- 21.3 Projeto Sala Limpa 496
 - 21.3.1 Refinamento de projeto 496
 - 21.3.2 Verificação de projeto 497
- 21.4 Teste Sala Limpa 498
 - 21.4.1 Teste estatístico de uso 498
 - 21.4.2 Certificação 499
- 21.5 Conceitos de Métodos Formais 500
- 21.6 Aplicando Notação Matemática para Especificação Formal 503
- 21.7 Linguagens de Especificação Formal 504
 - 21.7.1 *Object Constraint Language* (OCL) 505
 - 21.7.2 A linguagem de especificação Z 508
- 21.8 Resumo 510
- Problemas e Pontos a Ponderar 511
- Leituras e Fontes de Informação Complementares 512

CAPÍTULO 22 GESTÃO DE CONFIGURAÇÃO DE SOFTWARE 514

- 22.1 Gestão de Configuração de Software 515
 - 22.1.1 Um cenário SCM 515
 - 22.1.2 Elementos de um sistema de gestão de configuração 516
 - 22.1.3 Referenciais 517
 - 22.1.4 Itens de configuração de software 518
- 22.2 O Repositório SCM 519
 - 22.2.1 O papel do repositório 519
 - 22.2.2 Características gerais e conteúdo 519
 - 22.2.3 Características SCM 520
- 22.3 O Processo SCM 521
 - 22.3.1 Identificação de objetos na configuração de software 522
 - 22.3.2 Controle de versão 523
 - 22.3.3 Controle de alterações 524
 - 22.3.4 Auditoria de configuração 526
 - 22.3.5 Relatório de status 527
- 22.4 Gestão de Configuração para WebApps 528
 - 22.4.1 Problemas dominantes 528
 - 22.4.2 Objetos de configuração de WebApp 529
 - 22.4.3 Gestão de conteúdo 530
 - 22.4.4 Gestão de alterações 531
 - 22.4.5 Controle de versão 534
 - 22.4.6 Auditoria e relatório 534
- 22.5 Resumo 535
- Problemas e Pontos a Ponderar 536
- Leituras e Fontes de Informação Complementares 537

CAPÍTULO 23 MÉTRICAS DE PRODUTO 538

- 23.1 Estrutura para Métricas de Produto 539
 - 23.1.1 Medidas, métricas e indicadores 539

23.1.2	O desafio das métricas de produto	539
23.1.3	Princípios de medição	540
23.1.4	Medição de software orientada a objetivo	541
23.1.5	Atributos de métricas efetivas de software	542
23.2	Métricas para o Modelo de Requisitos	543
23.2.1	Métricas baseadas em função	543
23.2.2	Métricas para qualidade de especificação	546
23.3	Métricas para o Modelo de Projeto	547
23.3.1	Métricas de projeto da arquitetura	547
23.3.2	Métricas para projeto orientado a objeto	549
23.3.3	Métricas orientadas a classe — o conjunto de métricas CK	551
23.3.4	Métricas orientadas a classe – o conjunto de métricas MOOD	553
23.3.5	Métricas orientadas a objeto propostas por Lorenz e Kidd	554
23.3.6	Métricas de projeto em nível de componente	554
23.3.7	Métricas orientadas a operação	556
23.3.8	Métricas de projeto de interface de usuário	557
23.4	Métricas de Projeto para WebApps	557
23.5	Métricas para Código-Fonte	559
23.6	Métricas para Teste	560
23.6.1	Métricas de Halstead aplicadas ao teste	561
23.6.2	Métricas para teste orientado a objeto	561
23.7	Métricas para Manutenção	562
23.8	Resumo	563
	Problemas e Pontos a Ponderar	563
	Leituras e Fontes de Informação Complementares	564

PARTE QUATRO GERENCIAMENTO DE PROJETOS DE SOFTWARE 565

CAPÍTULO 24 CONCEITOS DE GERENCIAMENTO DE PROJETO 566

24.1	O Espectro de Gerenciamento	567
24.1.1	Pessoal	567
24.1.2	O produto	567
24.1.3	O processo	568
24.1.4	O projeto	568
24.2	Pessoas	568
24.2.1	Os interessados (comprometidos)	569
24.2.2	Líderes de equipe	569
24.2.3	Equipe de software	570
24.2.4	Equipes ágeis	572
24.2.5	Itens de comunicação e coordenação	573
24.3	O Produto	574
24.3.1	Escopo de software	574
24.3.2	Decomposição do problema	574
24.4	O Processo	575
24.4.1	Combinando o produto e o processo	575
24.4.2	Decomposição do processo	576
24.5	O Projeto	577
24.6	O Princípio W ⁵ HH	578
24.7	Práticas Vitais	579
24.8	Resumo	579
	Problemas e Pontos a Ponderar	580
	Leituras e Fontes de Informação Complementares	581

CAPÍTULO 25 MÉTRICAS DE PROCESSO E PROJETO 583

- 25.1 Métricas no domínio de processo e projeto 584
 - 25.1.1 Métricas de processo e aperfeiçoamento do processo de software 584
 - 25.1.2 Métricas de projeto 586
- 25.2 Medidas de Software 586
 - 25.2.1 Métricas orientadas a tamanho 588
 - 25.2.2 Métricas orientadas a função 589
 - 25.2.3 Reconciliando métricas LOC e FP 589
 - 25.2.4 Métricas orientadas a objeto 591
 - 25.2.5 Métricas orientadas a casos de uso 592
 - 25.2.6 Métricas de projeto WebApp 592
- 25.3 Métricas para Qualidade de Software 594
 - 25.3.1 Medição da qualidade 594
 - 25.3.2 Eficiência na remoção de defeitos 595
- 25.4 Integrando Métricas Dentro do Processo de Software 596
 - 25.4.1 Argumentos favoráveis a métricas de software 597
 - 25.4.2 Estabelecendo uma linha de base 597
 - 25.4.3 Coleta, cálculo e avaliação de métricas 597
- 25.5 Métricas para Pequenas Organizações 598
- 25.6 Estabelecendo um Programa de Métricas de Software 599
- 25.7 Resumo 601
- Problemas e Pontos a Ponderar 601
- Leituras e Fontes de Informação Complementares 602

CAPÍTULO 26 ESTIMATIVAS DE PROJETO DE SOFTWARE 604

- 26.1 Observações e Estimativas 605
- 26.2 O Processo de Planejamento do Projeto 606
- 26.3 Escopo e Viabilidade do Software 606
- 26.4 Recursos 607
 - 26.4.1 Recursos humanos 608
 - 26.4.2 Recursos de software reutilizáveis 608
 - 26.4.3 Recursos de ambiente 608
- 26.5 Estimativa do Projeto de Software 609
- 26.6 Técnicas de Decomposição 610
 - 26.6.1 Dimensionamento do software 610
 - 26.6.2 Estimativa baseada em problema 611
 - 26.6.3 Um exemplo de estimativa baseada em LOC 612
 - 26.6.4 Um exemplo de estimativa baseada em FP 613
 - 26.6.5 Estimativas baseadas em processo 614
 - 26.6.6 Um exemplo de estimativa baseada em processo 615
 - 26.6.7 Estimativa com casos de uso 616
 - 26.6.8 Exemplo de estimativa baseada em caso de uso 617
 - 26.6.9 Reconciliando estimativas 617
- 26.7 Modelos Empíricos de Estimativa 618
 - 26.7.1 Estrutura dos modelos de estimativa 619
 - 26.7.2 O modelo COCOMO II 619
 - 26.7.3 A equação do software 621
- 26.8 Estimativa para Projetos Orientados a Objeto 622
- 26.9 Técnicas Especializadas de Estimativa 622
 - 26.9.1 Estimativa para desenvolvimento ágil 622
 - 26.9.2 Estimativa para projetos para WebApp 623

26.10	A Decisão Fazer/Comprar	624
26.10.1	Criando uma árvore de decisões	624
26.10.2	Terceirização	625
26.11	Resumo	627
	Problemas e Pontos a Ponderar	627
	Leituras e Fontes de Informação Complementares	628

CAPÍTULO 27 CRONOGRAMA DE PROJETO 629

27.1	Conceitos Básicos	630
27.2	Cronograma de Projeto	631
27.2.1	Princípios básicos	632
27.2.2	Relação entre pessoas e esforço	632
27.2.3	Distribuição de esforço	634
27.3	Definindo um Conjunto de Tarefas para o Projeto de Software	635
27.3.1	Exemplo de conjunto de tarefas	635
27.3.2	Refinamento das ações de engenharia de software	636
27.4	Definindo uma Rede de Tarefas	636
27.5	Cronograma	637
27.5.1	Gráfico de Gantt	638
27.5.2	Acompanhando o cronograma	638
27.5.3	Acompanhando o progresso de um projeto orientado a objeto	640
27.5.4	Cronograma para projetos para WebApp	641
27.6	Análise de Valor Agregado	643
27.8	Resumo	645
	Problemas e Pontos a Ponderar	645
	Leituras e Fontes de Informação Complementares	646

CAPÍTULO 28 GESTÃO DE RISCOS 648

28.1	Estratégias de Riscos Reativa versus Proativa	649
28.2	Riscos de Software	649
28.3	Identificação do Risco	650
28.3.1	Avaliando o risco geral do projeto	651
28.3.2	Componentes e fatores de risco	651
28.4	Previsão de Risco	652
28.4.1	Desenvolvendo uma tabela de risco	653
28.4.2	Avaliando o impacto do risco	655
28.5	Refinamento do Risco	656
28.6	Mitigação, Monitoração e Controle de Riscos (RMMM)	657
28.7	O Plano RMMM	658
28.8	Resumo	660
	Problemas e Pontos a Ponderar	660
	Leituras e Fontes de Informação Complementares	661

CAPÍTULO 29 MANUTENÇÃO E REENGENHARIA 662

29.1	Manutenção de Software	663
29.2	Suportabilidade do Software	664
29.3	Reengenharia	665
29.4	Reengenharia de Processo de Negócio	665
29.4.1	Processos de negócio	665
29.4.2	Um modelo de BPR	666

29.5	Reengenharia de Software	667
29.5.1	Um modelo de processo de reengenharia de software	668
29.5.2	Atividades de reengenharia de software	669
29.6	Engenharia Reversa	670
29.6.1	Engenharia reversa para entender os dados	672
29.6.2	Engenharia reversa para entender o processamento	672
29.6.3	Engenharia reversa das interfaces de usuário	673
29.7	Reestruturação	674
29.7.1	Reestruturação de código	674
29.7.2	Reestruturação de dados	674
29.8	Engenharia Direta	675
29.8.1	Engenharia direta para arquiteturas cliente-servidor	676
29.8.2	Engenharia direta para arquiteturas orientadas a objeto	677
29.9	A Economia da Reengenharia	677
29.10	Resumo	678
	Problemas e Pontos a Ponderar	679
	Leituras e Fonte de Informação Complementares	679

PARTE CINCO **TÓPICOS AVANÇADOS** **681**

CAPÍTULO 30 **MELHORIA DO PROCESSO DE SOFTWARE** **682**

30.1	O Que É SPI?	683
30.1.1	Abordagens para SPI	683
30.1.2	Modelos de maturidade	685
30.1.3	A SPI é para todos?	685
30.2	O Processo de SPI	686
30.2.1	Avaliação e análise de lacunas	686
30.2.2	Educação e treinamento	687
30.2.3	Seleção e justificção	688
30.2.4	Instalação/migração	689
30.2.5	Mensuração	689
30.2.6	Gerenciamento de risco para SPI	689
30.2.7	Fatores críticos de sucesso	690
30.3	A CMMI	691
30.4	A CMM das Pessoas	695
30.5	Outras Estruturas SPI	696
30.6	Retorno sobre Investimento em SPI	697
30.7	Tendências da SPI	698
30.8	Resumo	698
	Problemas e Pontos a Ponderar	699
	Leituras e Fontes de Informação Complementares	699

CAPÍTULO 31 **TENDÊNCIAS EMERGENTES NA ENGENHARIA DE SOFTWARE** **701**

31.1	Evolução da Tecnologia	702
31.2	Observando Tendências na Engenharia de Software	703
31.3	Identificando as “Tendências Leves”	704
31.3.1	Administrando a complexidade	705
31.3.2	Software aberto	706
31.3.3	Requisitos emergentes	707
31.3.4	O mix de talentos	707
31.3.5	Blocos básicos de software	708

31.3.6	Mudando as percepções de “valor”	708
31.3.7	Código aberto	709
31.4	Direções da Tecnologia	710
31.4.1	Tendências de processo	710
31.4.2	O grande desafio	711
31.4.3	Desenvolvimento colaborativo	712
31.4.4	Engenharia de requisitos	713
31.4.5	Desenvolvimento de software controlado por modelo	714
31.4.6	Projeto pós-moderno	714
31.4.7	Desenvolvimento baseado em teste	715
31.5	Tendências Relacionadas com Ferramentas	716
31.5.1	Ferramentas que respondem a tendências leves	717
31.5.2	Ferramentas que lidam com as tendências tecnológicas	718
31.6	Resumo	719
	Problemas e Pontos a Ponderar	719
	Leituras e Fontes de Informação Complementares	720

CAPÍTULO 32 COMENTÁRIOS FINAIS 721

32.1	A Importância do Software – Revisitada	722
32.2	Profissionais e a Maneira como Constroem Sistemas	722
32.3	Novos Modos de Representar as Informações	723
32.4	A Visão no Longo Prazo	724
32.5	A Responsabilidade do Engenheiro de Software	725
32.6	Comentário Final	726

APÊNDICE 1 INTRODUÇÃO À UML 727**APÊNDICE 2 CONCEITOS ORIENTADOS AO OBJETO 744****REFERÊNCIAS 751****ÍNDICE 773**

SOFTWARE E
ENGENHARIA DE SOFTWARECONCEITOS-
-CHAVE

atividades de apoio	40
campos de aplicação	34
características de software	32
engenharia de software	38
metodologia	40
mitos de software. .	45
prática	45
princípios	44
processo de software	40
software legado . .	36
WebApps	37

Ele tinha a aparência clássica de um executivo sênior de uma grande empresa de software — cerca de 40 anos de idade, as têmporas levemente grisalhas, elegante e atlético, olhos penetrantes enquanto falava. Mas o que ele disse me deixou em choque: “O software está morto”.

Fiquei surpreso e então sorri. “Você está brincando, não é mesmo? O mundo é comandado pelo software e sua empresa tem lucrado imensamente com isso... Ele não está morto! Está vivo e crescendo.”

Balançando a cabeça enfática e negativamente, acrescentou: “Não, ele está morto... Pelo menos da forma que, um dia, o conhecemos”.

Assenti e disse: “Continue”.

Ele falava batendo na mesa para enfatizar: “A visão da velha escola de software — você o compra, é seu proprietário e é o responsável pelo seu gerenciamento — está chegando ao fim. Atualmente, com a Web 2.0 e a computação pervasiva, que vem surgindo com força, estamos por ver uma geração completamente diferente de software. Ele será distribuído via Internet e parecerá estar residente nos dispositivos do computador de cada usuário... Porém, estará residente em um servidor bem distante”.

PANORAMA

O que é? Software de computador é o produto que profissionais de software desenvolvem e ao qual dão suporte no longo prazo. Abrange programas executáveis em um computador de qualquer porte ou arquitetura, conteúdos (apresentados à medida que os programas são executados), informações descritivas tanto na forma impressa (*hard copy*) como na virtual, abrangendo praticamente qualquer mídia eletrônica. A engenharia de software abrange um processo, um conjunto de métodos (práticas) e um leque de ferramentas que possibilitam aos profissionais desenvolverem software de altíssima qualidade.

Quem realiza? Os engenheiros de software criam e dão suporte a ele e, praticamente, todos do mundo industrializado o utilizam, direta ou indiretamente.

Por que ele é importante? Software é importante porque afeta a quase todos os aspectos de nossas vidas e tornou-se pervasivo (incorporado) no comércio, na cultura e em nossas atividades cotidianas. A engenharia de software é importante

porque ela nos capacita para o desenvolvimento de sistemas complexos dentro do prazo e com alta qualidade.

Quais são as etapas envolvidas? Cria-se software para computadores da mesma forma que qualquer produto bem-sucedido: aplicando-se um processo adaptável e ágil que conduza a um resultado de alta qualidade, atendendo às necessidades daqueles que usarão o produto. Aplica-se uma abordagem de engenharia de software.

Qual é o artefato? Do ponto de vista de um engenheiro de software, é um conjunto de programas, conteúdo (dados) e outros artefatos que são software. Porém, do ponto de vista do usuário, o artefato consiste em informações resultantes que, de alguma forma, tornam a vida dele melhor.

Como garantir que o trabalho foi feito corretamente? Leia o restante deste livro, selecione as ideias que são aplicáveis ao software que você desenvolver e use-as em seu trabalho.

Eu tive de concordar: “Assim, sua vida será muito mais simples. Vocês, meus amigos, não terão de se preocupar com cinco versões diferentes do mesmo aplicativo em uso por dezenas de milhares de usuários espalhados”.

Ele sorriu e acrescentou: “Absolutamente. Apenas a versão mais atual residindo em nossos servidores. Quando fizermos uma modificação ou correção, forneceremos funcionalidade e conteúdo atualizados para todos os usuários. Todos terão isso instantaneamente!”.

Fiz uma careta: “Mas, da mesma forma, se houver um erro, todos o terão instantaneamente”.

Ele riu: “É verdade, por isso estamos redobrando nossos esforços para aplicarmos a engenharia de software de uma forma ainda melhor. O problema é que temos de fazer isso ‘rapidamente’, pois o mercado tem se acelerado, em todas as áreas de aplicação”.

Recostei-me e, com minhas mãos por trás da cabeça, disse: “Você sabe o que falam por aí: você pode ter algo rápido, você pode ter algo correto ou você pode ter algo barato. Escolha dois!”.

“Eu fico com rápido e correto”, disse, levantando-se.

Também me levantei, concluindo: “Então, nós realmente precisamos de engenharia de software”.

“Sei disso”, afirmou, enquanto se afastava. “O problema é: temos de convencer ainda outra geração de ‘techies’* de que isso é verdadeiro!”.

O software está *realmente* morto? Se estivesse, você não estaria lendo este livro!

Software de computadores continua a ser a tecnologia única mais importante no cenário mundial. E é também um ótimo exemplo da lei das consequências não intencionais. Há cinquenta anos, ninguém poderia prever que o software iria se tornar uma tecnologia indispensável para negócios, ciência e engenharia; que software iria viabilizar a criação de novas tecnologias (por exemplo, engenharia genética e nanotecnologia), a extensão de tecnologias existentes (por exemplo, telecomunicações) e a mudança radical nas tecnologias mais antigas (por exemplo, indústria gráfica); que software se tornaria a força motriz por trás da revolução do computador pessoal; que produtos de pacotes de software seriam comprados pelos consumidores em lojas de bairro; que software evoluiria lentamente de produto para serviço, na medida que empresas de software “sob encomenda” oferecessem funcionalidade imediata (*just-in-time*), via um navegador Web; que uma companhia de software iria se tornar a maior e mais influente do que quase todas as companhias da era industrial; que uma vasta rede comandada por software, denominada Internet, iria evoluir e modificar tudo: de pesquisa em bibliotecas a compras feitas pelos consumidores, incluindo discurso político, hábitos de namoros de jovens e de adultos não tão jovens.

Ninguém poderia prever que o software seria incorporado em sistemas de todas as áreas: transportes, medicina, telecomunicações, militar, industrial, entretenimento, máquinas de escritório... A lista é quase infindável. E se você acredita na lei das consequências não intencionais, há muitos efeitos que ainda não somos capazes de prever.

Ninguém poderia prever que milhões de programas de computador teriam de ser corrigidos, adaptados e ampliados à medida que o tempo passasse. A realização dessas atividades de “manutenção” absorve mais pessoas e recursos do que todo o esforço aplicado na criação de um novo software.

Conforme aumenta a importância do software, a comunidade da área tenta desenvolver tecnologias que tornem mais fácil, mais rápido e mais barato desenvolver e manter programas de computador de alta qualidade. Algumas dessas tecnologias são direcionadas a um campo de aplicação específico (por exemplo, projeto e implementação de sites); outras são focadas em um campo de tecnologia (por exemplo, sistemas orientados a objetos ou programação orientada a aspectos); e ainda outras são de bases amplas (por exemplo, sistemas operacionais como o

“Ideias e descobertas tecnológicas são as forças propulsoras do crescimento econômico.”

The Wall Street Journal

* N. de R.T.: fanáticos por tecnologia.

Linux). Entretanto, nós ainda temos de desenvolver uma tecnologia de software que faça tudo isso, sendo que a probabilidade de surgir tal tecnologia no futuro é pequena. Ainda assim, as pessoas apostam seus empregos, seu conforto, sua segurança, seu entretenimento, suas decisões e suas próprias vidas em software. Tomara que estejam certas.

Este livro apresenta uma estrutura que pode ser utilizada por aqueles que desenvolvem software — pessoas que devem fazê-lo corretamente. A estrutura abrange um processo, um conjunto de métodos e uma gama de ferramentas que chamaremos de *engenharia de software*.

1.1 A NATUREZA DO SOFTWARE

PONTO-CHAVE

Software é tanto um produto como um veículo que distribui um produto.

Hoje, o software assume um duplo papel. Ele é um produto e, ao mesmo tempo, o veículo para distribuir um produto. Como produto, fornece o potencial computacional representado pelo hardware ou, de forma mais abrangente, por uma rede de computadores que podem ser acessados por hardware local. Independentemente de residir em um celular ou operar dentro de um mainframe, software é um transformador de informações — produzindo, gerenciando, adquirindo, modificando, exibindo ou transmitindo informações que podem ser tão simples quanto um único bit ou tão complexas quanto uma apresentação multimídia derivada de dados obtidos de dezenas de fontes independentes. Como veículo de distribuição do produto, o software atua como a base para o controle do computador (sistemas operacionais), a comunicação de informações (redes) e a criação e o controle de outros programas (ferramentas de software e ambientes).

O software distribui o produto mais importante de nossa era — a *informação*. Ele transforma dados pessoais (por exemplo, transações financeiras de um indivíduo) de modo que possam ser mais úteis num determinado contexto; gerencia informações comerciais para aumentar a competitividade; fornece um portal para redes mundiais de informação (Internet) e os meios para obter informações sob todas as suas formas.

O papel desempenhado pelo software tem passado por grandes mudanças ao longo dos últimos cinquenta anos. Aperfeiçoamentos significativos no desempenho do hardware, mudanças profundas nas arquiteturas computacionais, vasto aumento na capacidade de memória e armazenamento, e uma ampla variedade de exóticas opções de entrada e saída, tudo isso resultou em sistemas computacionais mais sofisticados e complexos. Sofisticação e complexidade podem produzir resultados impressionantes quando um sistema é bem-sucedido, porém, também podem trazer enormes problemas para aqueles que precisam desenvolver sistemas robustos.

Atualmente, uma enorme indústria de software tornou-se fator dominante nas economias do mundo industrializado. Equipes de especialistas em software, cada qual concentrando-se numa parte da tecnologia necessária para distribuir uma aplicação complexa, substituíram o programador solitário de antigamente. Ainda assim, as questões levantadas por esse programador solitário continuam as mesmas feitas hoje, quando os modernos sistemas computacionais são desenvolvidos:¹

- Por que concluir um software leva tanto tempo?
- Por que os custos de desenvolvimento são tão altos?
- Por que não conseguimos encontrar todos os erros antes de entregarmos o software aos clientes?
- Por que gastamos tanto tempo e esforço mantendo programas existentes?
- Por que continuamos a ter dificuldade em medir o progresso enquanto o software está sendo desenvolvido e mantido?

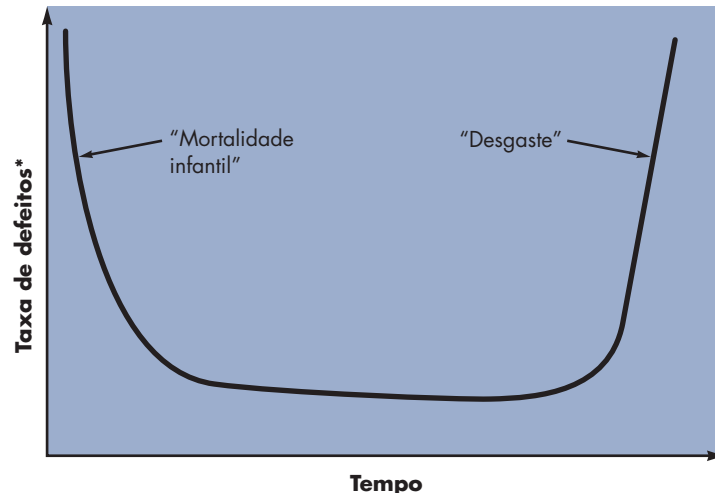
“Software é um lugar onde sonhos são plantados e pesadelos são colhidos, um pântano abstrato e místico onde demônios terríveis competem com mágicas panaceias, um mundo de lobisomens e balas de prata.”

Brad J. Cox

¹ Em um excelente livro de ensaio sobre o setor de software, Tom DeMarco [DeM95] contesta. Ele afirma: “Em vez de perguntar por que software custa tanto, precisamos começar perguntando: ‘O que fizemos para tornar possível que o software atual custe tão pouco?’ A resposta a essa pergunta nos ajudará a continuarmos com o extraordinário nível de realização que sempre tem distinguido a indústria de software”.

FIGURA 1.1

Curva de defeitos para hardware



Essas e muitas outras questões demonstram a preocupação com o software e a maneira como é desenvolvido — uma preocupação que tem levado à adoção da prática da engenharia de software.

1.1.1 Definindo software

Hoje em dia, a maior parte dos profissionais e muitos outros indivíduos do público em geral acham que entendem de software. Mas entendem mesmo?

Uma descrição de software em um livro-texto poderia ser a seguinte:

Software consiste em: (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso dos programas.

Sem dúvida, poderíamos dar outras definições mais completas.

Mas, provavelmente, uma definição mais formal não melhoraria, de forma considerável, a compreensão do que é software. Para conseguir isso, é importante examinar as características do software que o tornam diverso de outras coisas que os seres humanos constroem. Software é mais um elemento de sistema lógico do que físico. Dessa forma, tem características que são consideravelmente diferentes daquelas do hardware:

1. *Software é desenvolvido ou passa por um processo de engenharia; ele não é fabricado no sentido clássico.*

Embora existam algumas similaridades entre o desenvolvimento de software e a fabricação de hardware, as duas atividades são fundamentalmente diferentes. Em ambas, alta qualidade é obtida por meio de bom projeto, entretanto, a fase de fabricação de hardware pode gerar problemas de qualidade inexistentes (ou facilmente corrigíveis) para software. Ambas as atividades são dependentes de pessoas, mas a relação entre pessoas envolvidas e trabalho realizado é completamente diferente (ver Capítulo 24). Ambas requerem a construção de um “produto”, entretanto, as abordagens são diferentes. Os custos de software concentram-se no processo de engenharia. Isso significa que projetos de software não podem ser geridos como se fossem projetos de fabricação.

* N. de R.T.: os defeitos do software nem sempre se manifestam como falha, geralmente devido a tratamentos dos erros decorrentes destes defeitos pelo software. Esses conceitos serão precisamente mais detalhados e diferenciados nos capítulos sobre qualidade. Neste ponto, optou-se por traduzir *failure rate* por taxa de defeitos, sem prejuízo para a assimilação dos conceitos apresentados pelo autor neste capítulo.

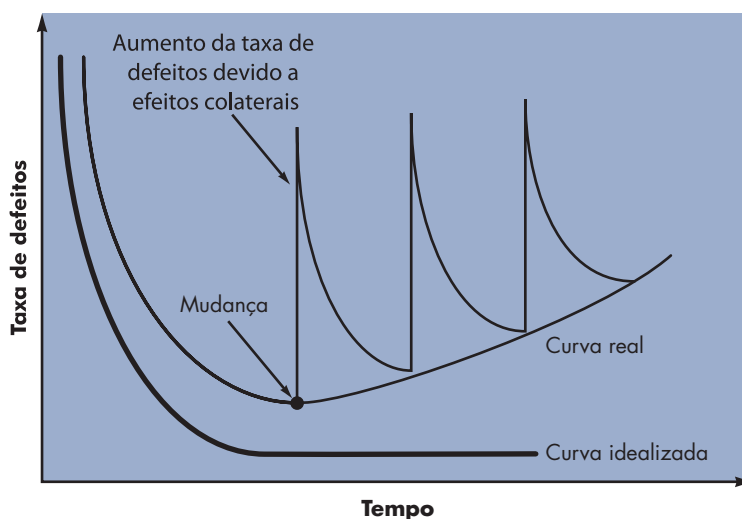
Como devemos definir software?

PONTO-CHAVE

Software é um processo de engenharia, não é fabricação.

FIGURA 1.2

Curvas de defeitos para software



PONTO-CHAVE

Software não se desgasta, mas ele se deteriora.

2. Software não “se desgasta”.

A Figura 1.1 representa a taxa de defeitos em função do tempo para hardware. Essa relação, normalmente denominada “curva da banheira”, indica que o hardware apresenta taxas de defeitos relativamente altas no início de sua vida (geralmente, atribuídas a defeitos de projeto ou de fabricação); os defeitos são corrigidos e a taxa cai para um nível estável (felizmente, bastante baixo) por certo período. Entretanto, à medida que o tempo passa, a taxa aumenta novamente, conforme os componentes de hardware sofrem os efeitos cumulativos de poeira, vibração, impactos, temperaturas extremas e vários outros males ambientais. Resumindo, o hardware começa a desgastar-se.

Software não é suscetível aos males ambientais que fazem com que o hardware se desgaste. Portanto, teoricamente, a curva da taxa de defeitos para software deveria assumir a forma da “curva idealizada”, mostrada na Figura 1.2. Defeitos ainda não descobertos irão resultar em altas taxas logo no início da vida de um programa. Entretanto, esses serão corrigidos e a curva se achata como mostrado. A curva idealizada é uma simplificação grosseira de modelos de defeitos reais para software. Porém, a implicação é clara: software não se desgasta, mas sim se *deteriora*!

Essa aparente contradição pode ser elucidada pela curva real apresentada na Figura 1.2. Durante sua vida², o software passará por alterações. À medida que estas ocorrerem, é provável que sejam introduzidos erros, fazendo com que a curva de taxa de defeitos se acentue, conforme mostrado na “curva real” (Figura 1.2). Antes que a curva possa retornar à taxa estável original, outra alteração é requisitada, fazendo com que a curva se acentue novamente. Lentamente, o nível mínimo da taxa começa a aumentar — o software está deteriorando devido à modificação.

Outro aspecto de desgaste ilustra a diferença entre hardware e software. Quando um componente de hardware se desgasta, ele é substituído por uma peça de reposição. Não existem peças de reposição de software. Cada defeito de software indica um erro no projeto ou no processo pelo qual o projeto foi traduzido em código de máquina executável. Portanto, as tarefas de manutenção de software, que envolvem solicitações de mudanças, implicam em complexidade consideravelmente maior do que a de manutenção de hardware.



Caso queira reduzir a deterioração do software, terá de fazer um projeto melhor de software (Capítulos 8 a 13).

PONTO-CHAVE

Os métodos de engenharia de software tentam reduzir ao máximo a magnitude das elevações (picos) e a inclinação da curva real da Figura 1.2.

² De fato, desde o momento em que o desenvolvimento começa, e muito antes da primeira versão ser entregue, podem ser solicitadas mudanças por uma variedade de diferentes interessados.

"Ideias são a matéria-prima para construção de ideias."

Jason Zebheazy

3. *Embora a indústria caminhe para a construção com base em componentes, a maioria dos softwares continua a ser construída de forma personalizada (sob encomenda).*

À medida que a disciplina da engenharia evolui, uma coleção de componentes de projeto padronizados é criada. Parafusos padronizados e circuitos integrados de linha são apenas dois dos milhares de componentes padronizados utilizados por engenheiros mecânicos e elétricos ao projetarem novos sistemas. Os componentes reutilizáveis foram criados para que o engenheiro possa se concentrar nos elementos realmente inovadores de um projeto, isto é, nas partes do projeto que representam algo novo. No mundo do hardware, a reutilização de componentes é uma parte natural do processo de engenharia. No mundo do software, é algo que, em larga escala, apenas começou a ser alcançado.

Um componente de software deve ser projetado e implementado de modo que possa ser reutilizado em muitos programas diferentes. Os modernos componentes reutilizáveis encapsulam tanto dados quanto o processamento aplicado a eles, possibilitando criar novas aplicações a partir de partes reutilizáveis.³ Por exemplo, as atuais interfaces interativas com o usuário são construídas com componentes reutilizáveis que possibilitam criar janelas gráficas, menus "pull-down" (suspensos e retráteis) e uma ampla variedade de mecanismos de interação. Estruturas de dados e detalhes de processamento necessários para a construção da interface ficam em uma biblioteca de componentes reutilizáveis para a construção de interfaces.

1.1.2 Campos de aplicação de software

Hoje em dia, sete grandes categorias de software apresentam desafios contínuos para os engenheiros de software:

Software de sistema — conjunto de programas feito para atender a outros programas. Certos softwares de sistema (por exemplo, compiladores, editores e utilitários para gerenciamento de arquivos) processam estruturas de informação complexas, porém, determinadas.⁴ Outras aplicações de sistema (por exemplo, componentes de sistema operacional, drivers, software de rede, processadores de telecomunicações) processam dados amplamente indeterminados. Em ambos os casos, a área de software de sistemas é caracterizada por "pesada" interação com o hardware do computador; uso intenso por múltiplos usuários; operação concorrente que requer escala da ordem, compartilhamento de recursos e gestão de processo sofisticada; estruturas de dados complexas e múltiplas interfaces externas.

Software de aplicação — programas sob medida que solucionam uma necessidade específica de negócio. Aplicações nessa área processam dados comerciais ou técnicos de uma forma que facilite operações comerciais ou tomadas de decisão administrativas/técnicas. Além das aplicações convencionais de processamento de dados, o software de aplicação é usado para controlar funções de negócio em tempo real (por exemplo, processamento de transações em pontos de venda, controle de processos de fabricação em tempo real).

Software científico/de engenharia — tem sido caracterizado por algoritmos "number crunching" (para "processamento numérico pesado"). As aplicações vão da astronomia à vulcanologia, da análise de tensões na indústria automotiva à dinâmica orbital de ônibus espaciais, e da biologia molecular à fabricação automatizada. Entretanto, aplicações modernas dentro da área de engenharia/científica estão se afastando dos algoritmos numéricos convencionais. Projeto com o auxílio de computador, simulação de sistemas e outras aplicações interativas começaram a ter características de sistemas em tempo real e até mesmo de software de sistemas.

WebRef

Uma das mais abrangentes bibliotecas de shareware/freeware (software compartilhado/livre) pode ser encontrada em shareware.cnet.com.

³ O desenvolvimento com base em componentes é discutido no Capítulo 10.

⁴ Um software é determinado se a ordem e o *timing* (periodicidade, frequência, medidas de tempo) de entradas, processamento e saídas forem previsíveis. É indeterminado, se ordem e *timing* de entradas, processamento e saídas não puderem ser previstos antecipadamente.

Software embutido — residente num produto ou sistema e utilizado para implementar e controlar características e funções para o usuário final e para o próprio sistema. Executa funções limitadas e específicas (por exemplo, controle do painel de um forno micro-ondas) ou fornece função significativa e capacidade de controle (por exemplo, funções digitais de automóveis, tal como controle do nível de combustível, painéis de controle e sistemas de freios).

Software para linha de produtos — projetado para prover capacidade específica de utilização por muitos clientes diferentes. Pode focalizar um mercado limitado e particularizado (por exemplo, produtos para controle de estoques) ou direcionar-se para mercados de consumo de massa (por exemplo, processamento de texto, planilhas eletrônicas, computação gráfica, multimídia, entretenimento, gerenciamento de bancos de dados e aplicações financeiras pessoais e comerciais).

Aplicações para a Web — chamadas de “WebApps”, essa categoria de software centralizada em redes abarca uma vasta gama de aplicações. Em sua forma mais simples, as WebApps podem ser pouco mais que um conjunto de arquivos de hipertexto interconectados, apresentando informações por meio de texto e informações gráficas limitadas. Entretanto, com o aparecimento da Web 2.0, elas têm evoluído e se transformado em sofisticados ambientes computacionais que não apenas fornecem recursos especializados, funções computacionais e conteúdo para o usuário final, como também estão integradas a bancos de dados corporativos e aplicações comerciais.

Software de inteligência artificial — faz uso de algoritmos não numéricos para solucionar problemas complexos que não são passíveis de computação ou de análise direta. Aplicações nessa área incluem: robótica, sistemas especialistas, reconhecimento de padrões (de imagem e de voz), redes neurais artificiais, prova de teoremas e jogos.

Milhões de engenheiros de software em todo o mundo trabalham arduamente em projetos de software em uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas em muitos outros, aplicações já existentes estão sendo corrigidas, adaptadas e aperfeiçoadas. Não é incomum para um jovem engenheiro de software trabalhar num programa mais velho que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma das categorias discutidas. Espera-se que o legado a ser deixado por essa geração facilite o trabalho de futuros engenheiros de software. Ainda assim, novos desafios (Capítulo 31) têm surgido no horizonte:

Computação mundial aberta — o rápido crescimento de redes sem fio pode, em breve, conduzir a uma verdadeira computação distribuída e pervasiva (ampliada, compartilhada e incorporada nos ambientes domésticos e comerciais). O desafio para os engenheiros de software será o de desenvolver sistemas e software aplicativo que permitam que dispositivos móveis, computadores pessoais e sistemas corporativos se comuniquem através de extensas redes.

Netsourcing (recursos via Internet) — a Internet está se tornando, rapidamente, tanto um mecanismo computacional, como um provedor de conteúdo. O desafio para os engenheiros de software consiste em arquitetar aplicações simples (isto é, planejamento financeiro pessoal) e sofisticadas que forneçam benefícios aos mercados mundiais de usuários finais visados.

Software aberto — uma tendência crescente que resulta na distribuição de código-fonte para aplicações de sistemas (por exemplo, sistemas operacionais, bancos de dados e ambientes de desenvolvimento), de forma que muitas pessoas possam contribuir para seu desenvolvimento. O desafio para os engenheiros de software consiste em construir um código-fonte autodescritivo, porém, mais importante ainda, será desenvolver técnicas que permitam que tanto clientes quanto desenvolvedores saibam quais alterações foram feitas e como se manifestam dentro do software.

“Não existe
computador que
tenha bom senso.”

Marvin Minsky

“Você não pode sempre prever, mas pode sempre se preparar.”

Anônimo

Cada um desses desafios obedecerá, sem dúvida, à lei das consequências não intencionais, produzindo efeitos (para executivos, engenheiros de software e usuários finais) que, hoje, não podem ser previstos. Entretanto, os engenheiros de software podem se preparar, iniciando um processo que seja ágil e suficientemente adaptável para assimilar as profundas mudanças na tecnologia e nas regras comerciais, que certamente virão na próxima década.

1.1.3 Software legado

Centenas de milhares de programas de computador caem em um dos sete amplos campos de aplicação discutidos na subseção anterior. Alguns deles são software de ponta — recém-lançados para indivíduos, indústria e governo. Outros programas são mais antigos, em alguns casos *muito* mais antigos.

Esses programas mais antigos — frequentemente denominados *software legado* — têm sido foco de contínua atenção e preocupação desde os anos 1960. Dayani-Fard e seus colegas [Day99] descrevem software legado da seguinte maneira:

Sistemas de software legado... Foram desenvolvidos décadas atrás e têm sido continuamente modificados para se adequar a mudanças dos requisitos de negócio e a plataformas computacionais. A proliferação de tais sistemas está causando dores de cabeça para grandes organizações que os consideram dispendiosos de manter e arriscados de evoluir.

Liu e seus colegas [Liu98] ampliam essa descrição observando que “muitos sistemas legados permanecem dando suporte para funções de negócio vitais e são ‘indispensáveis’ para o mesmo”. Por isso, um software legado é caracterizado pela longevidade e criticidade de negócios.

Infelizmente, algumas vezes, há uma característica adicional que pode estar presente em um software legado: *baixa qualidade*.⁵ Os sistemas legados, algumas vezes, têm projetos não expansíveis, código intrincado, documentação pobre ou inexistente, casos de testes e resultados que nunca foram arquivados, um histórico de modificações mal administrado — a lista pode ser bem longa. Ainda assim, esses sistemas dão suporte a “funções vitais de negócio e são indispensáveis para ele”. O que fazer então?

A única resposta razoável talvez seja: *Não faça nada*, pelo menos até que o sistema legado tenha que passar por alguma modificação significativa. Se o software legado atende às necessidades de seus usuários e roda de forma confiável, ele não está “quebrado” e não precisa ser “consertado”. Entretanto, com o passar do tempo, esses sistemas, frequentemente, evoluem por uma ou mais das seguintes razões:

- O software deve ser adaptado para atender às necessidades de novos ambientes ou de novas tecnologias computacionais.
- O software deve ser aperfeiçoado para implementar novos requisitos de negócio.
- O software deve ser expandido para torná-lo interoperável com outros bancos de dados ou com sistemas mais modernos.
- O software deve ser rearquitetado para torná-lo viável dentro de um ambiente de rede.

Quando essas modalidades de evolução ocorrerem, um sistema legado deve passar por re-engenharia (Capítulo 29) para que permaneça viável no futuro. O objetivo da engenharia de software moderna é o de “elaborar metodologias baseadas na noção de evolução”; isto é, na noção de que os sistemas de software modificam-se continuamente, novos sistemas são construídos a partir dos antigos e... Todos devem interoperar e cooperar um com o outro” [Day99].

 O que faço caso encontre um sistema legado de baixa qualidade?

 Que tipos de mudanças são feitas em sistemas legados?



Todo engenheiro de software deve reconhecer que modificações são naturais. Não tente combatê-las.

⁵ Nesse caso, a qualidade é julgada pensando-se em termos de engenharia de software moderna — um critério um tanto injusto, já que alguns conceitos e princípios da engenharia de software moderna talvez não tenham sido bem entendidos na época em que o software legado foi desenvolvido.

1.2 A NATUREZA ÚNICA DAS WEBAPPS

“Quando notarmos qualquer tipo de estabilidade, a Web terá se transformado em algo completamente diferente.”

Louis Monier

Nos primórdios da World Wide Web (por volta de 1990 a 1995), os sites eram formados por nada mais que um conjunto de arquivos de hipertexto ligados que apresentavam informações usando texto e gráficos limitados. Com o tempo, o aumento da HTML, via ferramentas de desenvolvimento (por exemplo, XML, Java), tornou possível aos engenheiros da Internet oferecerem capacidade computacional juntamente com as informações. Nasceram, então, os *sistemas e aplicações baseados na Web*⁶ (refiro-me a eles coletivamente como aplicações *WebApps*). Atualmente, as *WebApps* evoluíram para sofisticadas ferramentas computacionais que não apenas oferecem funções especializadas (*stand-alone functions*) ao usuário final, como também foram integradas aos bancos de dados corporativos e às aplicações de negócio.

Conforme observado na Seção 1.1.2, *WebApps* são apenas uma dentre uma série de diferentes categorias de software. Ainda assim, pode-se afirmar que elas são diferentes. Powell [Pow98] sugere que sistemas e aplicações baseados na Web “envolvem uma mistura de publicação impressa e desenvolvimento de software, de marketing e computação, de comunicações internas e relações externas e de arte e tecnologia”. Os seguintes atributos são encontrados na grande maioria das *WebApps*:

 **Que características diferenciam as WebApps de outros softwares?**

Uso intensivo de redes. Uma *WebApp* reside em uma rede e deve atender às necessidades de uma comunidade diversificada de clientes. A rede possibilita acesso e comunicação mundiais (isto é, a Internet) ou acesso e comunicação mais limitados (por exemplo, uma Intranet corporativa).

Simultaneidade. Um grande número de usuários pode acessar a *WebApp* ao mesmo tempo. Em muitos casos, os padrões de utilização entre os usuários finais variam amplamente.

Carga não previsível. O número de usuários da *WebApp* pode variar, em ordem de grandeza, de um dia para outro. Uma centena de usuários pode conectar-se na segunda-feira e 10.000 na quinta.

Desempenho. Se um usuário de uma *WebApp* tiver de esperar muito (para acesso, processamento no servidor, formatação e exibição no cliente), talvez ele procure outra opção.

Disponibilidade. Embora a expectativa de 100% de disponibilidade não seja razoável, usuários de *WebApps* populares normalmente exigem acesso 24 horas por dia, 7 dias por semana, 365 dias por ano. Usuários na Austrália ou Ásia podem requerer acesso quando aplicações de software domésticas tradicionais na América do Norte estejam off-line para manutenção.

Orientadas a dados. A função principal de muitas *WebApps* é usar hipermídias para apresentar texto, gráficos, áudio e vídeo para o usuário final. Além disso, as *WebApps* são comumente utilizadas para acessar informações em bancos de dados que não são parte integrante do ambiente baseado na Web (por exemplo, comércio eletrônico e/ou aplicações financeiras).

Sensibilidade no conteúdo. A qualidade e a natureza estética do conteúdo são fatores importantes que determinam a qualidade de uma *WebApp*.

Evolução contínua. Diferentemente de softwares de aplicação convencionais que evoluem ao longo de uma série de versões planejadas e cronologicamente espaçadas, as *WebApps* evoluem continuamente. Não é incomum algumas delas (especificamente seu conteúdo) serem atualizadas segundo uma escala minuto a minuto ou seu conteúdo ser computado independentemente para cada solicitação.

6 No contexto deste livro, o termo *aplicação Web* (*WebApp*) engloba tudo, de uma simples página Web que possa ajudar um consumidor a processar o pagamento do aluguel de um automóvel a um amplo site que fornece serviços de viagem completos para executivos e turistas. Dentro dessa categoria, estão sites completos, funcionalidade especializada dentro de sites e aplicações para processamento de informações residentes na Internet ou em uma Intranet ou Extranet.

Imediatismo. Embora *imediatismo* — a imperativa necessidade de colocar rapidamente um software no mercado — seja uma característica de diversos campos de aplicação, as WebApps normalmente apresentam um tempo de colocação no mercado que pode consistir de poucos dias ou semanas.⁷

Segurança. Pelo fato de estarem disponíveis via acesso à Internet, torna-se difícil, se não impossível, limitar o número dos usuários finais que podem acessar as WebApps. A fim de proteger conteúdos sensíveis e oferecer modos seguros de transmissão de dados, fortes medidas de segurança devem ser implementadas ao longo da infraestrutura que suporta uma WebApp e dentro da própria aplicação.

Estética. Parte inegável do apelo de uma WebApp consiste na sua aparência e na impressão que desperta. Quando uma aplicação for desenvolvida para o mercado ou para vender produtos ou ideias, a estética pode ser tão importante para o sucesso quanto o projeto técnico.

Pode-se argumentar que outras categorias de aplicação discutidas na Seção 1.1.2 podem exibir alguns dos atributos citados. Entretanto, as WebApps quase sempre apresentam todos esses atributos.

1.3 ENGENHARIA DE SOFTWARE

Para desenvolver software que esteja preparado para enfrentar os desafios do século vinte e um, devemos perceber uns poucos fatos reais:

PONTO-CHAVE

Entenda o problema antes de elaborar uma solução.

PONTO-CHAVE

Projetar é uma atividade fundamental na engenharia de software.

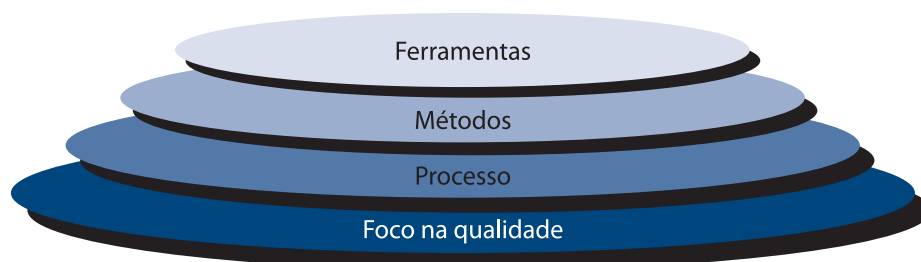
PONTO-CHAVE

Qualidade e facilidade de manutenção são resultantes de um projeto bem feito.

- Software tornou-se profundamente incorporado em praticamente todos os aspectos de nossas vidas e, conseqüentemente, o número de pessoas interessadas nos recursos e nas funções oferecidas por uma determinada aplicação⁸ tem crescido significativamente. Quando uma aplicação ou um sistema embutido estão para ser desenvolvidos, muitas vozes devem ser ouvidas. E, algumas vezes, parece que cada uma delas possui uma ideia ligeiramente diferente de quais funções ou recursos o software deve oferecer. Depreende-se, portanto, que se deve fazer *um esforço concentrado para compreender o problema antes de desenvolver uma solução de software*.
- Os requisitos de tecnologia de informação demandados por indivíduos, empresas e órgãos governamentais estão se tornando cada vez mais complexos a cada ano. Atualmente, equipes numericamente grandes desenvolvem programas de computador que antigamente eram desenvolvidos por um único indivíduo. Software sofisticado, outrora implementado em um ambiente computacional independente e previsível, hoje em dia está incorporado em tudo, de produtos eletrônicos de consumo a equipamentos médicos e sistemas de armamentos. A complexidade desses novos produtos e sistemas baseados em computadores demanda uma maior atenção para com as interações de todos os elementos do sistema. Depreende-se, portanto, que *projetar tornou-se uma atividade-chave (fundamental)*.
- Indivíduos, negócios e governos dependem, de forma crescente, de software para decisões estratégicas e táticas, assim como para controle e para operações cotidianas. Se o software falhar, as pessoas e as principais empresas poderão vivenciar desde pequenos inconvenientes a falhas catastróficas. Depreende-se, portanto, que *um software deve apresentar qualidade elevada*.
- À medida que o valor de uma aplicação específica aumente, a probabilidade é de que sua base de usuários e longevidade também cresçam. À medida que sua base de usuários e seu tempo em uso forem aumentando, a demanda por adaptação e aperfeiçoamento também irá aumentar. Conclui-se, portanto, que *um software deve ser passível de manutenção*.

⁷ Com o uso de ferramentas modernas, sofisticadas, páginas de sites podem ser produzidas em apenas algumas horas.

⁸ Neste livro, mais à frente, chamarei tais pessoas de “interessados”.

FIGURA 1.3**Camadas da engenharia de software**

Essas simples constatações nos conduzem a uma única conclusão: *software, em todas as suas formas e em todos os seus campos de aplicação, deve passar pelos processos de engenharia*. E isso nos leva ao tópico principal deste livro — *engenharia de software*.

Embora centenas de autores tenham criado suas definições pessoais de engenharia de software, uma definição proposta por Fritz Bauer [Nau69] na conferência sobre o tema serve de base para discussão:

[Engenharia de software é] o estabelecimento e o emprego de sólidos princípios de engenharia de modo a obter software de maneira econômica, que seja confiável e funcione de forma eficiente em máquinas reais.

Ficamos tentados a acrescentar algo a essa definição.⁹ Ela diz pouco sobre os aspectos técnicos da qualidade de software; ela não trata diretamente da necessidade de satisfação do cliente ou da entrega do produto dentro do prazo; ela não faz menção à importância das medições e métricas; ela não declara a importância de um processo eficaz. Ainda assim, a definição de Bauer nos fornece uma base. Quais são os “sólidos princípios de engenharia” que podem ser aplicados ao desenvolvimento de software? Como criar software “economicamente viável” e de modo “confiável”? O que é necessário para desenvolver programas de computador que funcionem “de forma eficiente” não apenas em uma, mas sim em várias e diferentes “máquinas reais”? Essas são as questões que continuam a desafiar os engenheiros de software.

A IEEE [IEE93a] desenvolveu uma definição mais abrangente ao afirmar o seguinte:

Engenharia de software: (1) A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software. (2) O estudo de abordagens como definido em (1).

Entretanto, uma abordagem “sistemática, disciplinada e quantificável”, aplicada por uma equipe de desenvolvimento de software, pode ser pesada para outra. Precisamos de disciplina, mas também precisamos de adaptabilidade e agilidade.

A engenharia de software é uma tecnologia em camadas. Referindo-se à Figura 1.3, qualquer abordagem de engenharia (inclusive engenharia de software) deve estar fundamentada em um comprometimento organizacional com a qualidade. A gestão da qualidade total Seis Sigma e filosofias similares¹⁰ promovem uma cultura de aperfeiçoamento contínuo de processos, e é esta cultura que, no final das contas, leva ao desenvolvimento de abordagens cada vez mais efetivas na engenharia de software. A pedra fundamental que sustenta a engenharia de software é o foco na qualidade.

A base para a engenharia de software é a camada de *processos*. O processo de engenharia de software é a liga que mantém as camadas de tecnologia coesas e possibilita o desenvolvimento de software de forma racional e dentro do prazo. O processo define uma metodologia que deve ser estabelecida para a entrega efetiva de tecnologia de engenharia de software. O processo de

“Mais do que uma simples disciplina ou ramo do conhecimento, *engineering* é um verbo [engenhar, engendrar], uma palavra de ação, uma maneira de abordar um problema.”

Scott Whitmir

? Como definimos engenharia de software?

PONTO-CHAVE

A engenharia de software engloba um processo, métodos de gerenciamento e desenvolvimento de software, bem como ferramentas.

⁹ Para inúmeras definições adicionais de *engenharia de software*, consulte: www.answers.com/topic/software-engineering#wp_note-13.

¹⁰ A gestão da qualidade e metodologias relacionadas são discutidas no Capítulo 14 e ao longo da Parte 3 deste livro.

WebRef

CrossTalk é um jornal que divulga informações práticas a respeito de processo, métodos e ferramentas. Pode ser encontrado no endereço: **www.stsc.hill.af.mil**.

software constitui a base para o controle do gerenciamento de projetos de software e estabelece o contexto no qual são aplicados métodos técnicos, são produzidos produtos derivados (modelos, documentos, dados, relatórios, formulários etc.), são estabelecidos marcos, a qualidade é garantida e mudanças são geridas de forma apropriada.

Os *métodos* da engenharia de software fornecem as informações técnicas para desenvolver software. Os métodos envolvem uma ampla gama de tarefas, que incluem: comunicação, análise de requisitos, modelagem de projeto, construção de programa, testes e suporte.

Os métodos da engenharia de software baseiam-se em um conjunto de princípios básicos que governam cada área da tecnologia e inclui atividades de modelagem e outras técnicas descritivas.

As *ferramentas* da engenharia de software fornecem suporte automatizado ou semiautomatizado para o processo e para os métodos. Quando as ferramentas são integradas, de modo que as informações criadas por uma ferramenta possam ser usadas por outra, é estabelecido um sistema para o suporte ao desenvolvimento de software, denominado *engenharia de software com o auxílio do computador*.

1.4 O PROCESSO DE SOFTWARE

? Quais são os elementos de um processo de software?

“Um processo define quem está fazendo o quê, quando e como para atingir determinado objetivo.”

Ivar Jacobson,
Grady Booch
e James Rumbaugh

Processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum produto de trabalho (*work product*). Uma *atividade* esforça-se para atingir um objetivo amplo (por exemplo, comunicar-se com os interessados) e é utilizada independentemente do campo de aplicação, do tamanho do projeto, da complexidade de esforços ou do grau de rigor com que a engenharia de software será aplicada. Uma *ação* (por exemplo, projeto de arquitetura) envolve um conjunto de tarefas que resultam num artefato de software fundamental (por exemplo, um modelo de projeto de arquitetura). Uma *tarefa* se concentra em um objetivo pequeno, porém, bem definido (por exemplo, realizar um teste de unidades) e produz um resultado tangível.

No contexto da engenharia de software, um processo *não* é uma prescrição rígida de como desenvolver um software. Ao contrário, é uma abordagem adaptável que possibilita às pessoas (a equipe de software) realizar o trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas. A intenção é a de sempre entregar software dentro do prazo e com qualidade suficiente para satisfazer àqueles que patrocinaram sua criação e àqueles que irão utilizá-lo.

Uma *metodologia* (*framework*) de processo estabelece o alicerce para um processo de engenharia de software completo, por meio da identificação de um pequeno número de *atividades estruturais* aplicáveis a todos os projetos de software, independentemente de tamanho ou complexidade. Além disso, a metodologia de processo engloba um conjunto de *atividades de apoio* (*umbrella activities* — *abertas*) aplicáveis em todo o processo de software. Uma metodologia de processo genérica para engenharia de software compreende cinco atividades:

Comunicação. Antes de iniciar qualquer trabalho técnico, é de vital importância comunicar-se e colaborar com o cliente (e outros interessados)¹¹. A intenção é compreender os objetivos das partes interessadas para com o projeto e fazer o levantamento das necessidades que ajudarão a definir as funções e características do software.

Planejamento. Qualquer jornada complicada pode ser simplificada caso exista um mapa. Um projeto de software é uma jornada complicada, e a atividade de planejamento cria um “mapa” que ajuda a guiar a equipe na sua jornada. O mapa — denominado *plano de projeto de software* — define o trabalho de engenharia de software, descrevendo as tarefas técnicas a ser

? Quais são as cinco atividades genéricas de metodologia de processo?

¹¹ *Interessado* é qualquer um que tenha interesse no êxito de um projeto — executivos, usuários finais, engenheiros de software, o pessoal de suporte etc. Rob Thomsett ironiza: “Interessado [*stakeholder*, em inglês] é uma pessoa que está segurando [*holding*, em inglês] uma estaca [*stake*, em inglês] grande e pontiaguda. Se você não cuidar de seus interessados, você sabe exatamente onde irá parar essa estaca.”

"Einstein argumentou que devia haver uma explicação simplificada da natureza, pois Deus não é caprichoso ou arbitrário. Tal fé não conforta o engenheiro de software. Grande parte da complexidade com a qual terá de lidar é arbitrária."

Fred Brooks

conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos resultantes a ser produzidos e um cronograma de trabalho.

Modelagem. Independentemente de ser um paisagista, um construtor de pontes, um engenheiro aeronáutico, um carpinteiro ou um arquiteto, trabalha-se com modelos todos os dias. Cria-se um "esboço" da coisa, de modo que se possa ter uma ideia do todo — qual será o seu aspecto em termos de arquitetura, como as partes constituintes se encaixarão e várias outras características. Se necessário, refina-se o esboço com mais detalhes, numa tentativa de compreender melhor o problema e como resolvê-lo. Um engenheiro de software faz a mesma coisa criando modelos para melhor entender as necessidades do software e o projeto que irá atender a essas necessidades.

Construção. Essa atividade combina geração de código (manual ou automatizada) e testes necessários para revelar erros na codificação.

Emprego. O software (como uma entidade completa ou como um incremento parcialmente efetivado) é entregue ao cliente, que avalia o produto entregue e fornece feedback, baseado na avaliação.

Essas cinco atividades metodológicas genéricas podem ser utilizadas para o desenvolvimento de programas pequenos e simples, para a criação de grandes aplicações para a Internet e para a engenharia de grandes e complexos sistemas baseados em computador. Os detalhes do processo de software serão bem diferentes em cada um dos casos, mas as atividades metodológicas permanecerão as mesmas.

Para muitos projetos de software, as atividades metodológicas são aplicadas iterativamente conforme o projeto se desenvolve. Ou seja, **comunicação, planejamento, modelagem, construção e emprego** são aplicados repetidamente quantas forem as iterações do projeto, sendo que cada iteração produzirá um *incremento de software*. Este disponibilizará uma parte dos recursos e funcionalidades do software. A cada incremento, o software torna-se mais e mais completo.

As atividades metodológicas do processo de engenharia de software são complementadas por uma série de *atividades de apoio*; em geral, estas são aplicadas ao longo de um projeto, ajudando a equipe a gerenciar, a controlar o progresso, a qualidade, as mudanças e o risco. As atividades de apoio típicas são:

Controle e acompanhamento do projeto — possibilita que a equipe avalie o progresso em relação ao plano do projeto e tome as medidas necessárias para cumprir o cronograma.

Administração de riscos — avalia riscos que possam afetar o resultado ou a qualidade do produto/projeto.

Garantia da qualidade de software — define e conduz as atividades que garantem a qualidade do software.

Revisões técnicas — avaliam artefatos da engenharia de software, tentando identificar e eliminar erros antes que se propaguem para a atividade seguinte.

Medição — define e coleta medidas (do processo, do projeto e do produto). Auxilia na entrega do software de acordo com os requisitos; pode ser usada com as demais atividades (metodológicas e de apoio).

Gerenciamento da configuração de software — gerencia os efeitos das mudanças ao longo do processo.

Gerenciamento da reusabilidade — define critérios para o reúso de artefatos (inclusive componentes de software) e estabelece mecanismos para a obtenção de componentes reutilizáveis.

PONTO-CHAVE

Atividades de apoio ocorrem ao longo do processo de software e se concentram, principalmente, no gerenciamento, acompanhamento e controle do projeto.

PONTO-CHAVE

A adaptação do processo de software é essencial para o sucesso de um projeto.

Preparo e produção de artefatos de software — engloba as atividades necessárias para criar artefatos como, por exemplo, modelos, documentos, logs, formulários e listas.

Cada uma dessas atividades universais será discutida de forma aprofundada mais adiante.

Anteriormente, declarei que o processo de engenharia de software não é rígido nem deve ser seguido à risca. Mais que isso, ele deve ser ágil e adaptável (ao problema, ao projeto, à equipe e à cultura organizacional). Portanto, o processo adotado para um determinado projeto pode ser muito diferente daquele adotado para outro. Entre as diferenças, temos:

- fluxo geral de atividades, ações e tarefas e suas interdependências;
- grau pelo qual ações e tarefas são definidas dentro de cada atividade da metodologia;
- grau pelo qual artefatos de software são identificados e exigidos;
- modo de aplicar as atividades de garantia de qualidade;
- modo de aplicar as atividades de acompanhamento e controle do projeto;
- grau geral de detalhamento e rigor da descrição do processo;
- grau de envolvimento com o projeto (por parte do cliente e de outros interessados);
- nível de autonomia dada à equipe de software;
- grau de prescrição da organização da equipe.

A Parte I deste livro examina o processo de software com grau de detalhamento considerável. Os *modelos de processo prescritivo* (Capítulo 2) abordam detalhadamente a definição, a identificação e a aplicação de atividades e tarefas do processo. A intenção é melhorar a qualidade do sistema, tornar os projetos mais gerenciáveis, tornar as datas de entrega e os custos mais previsíveis e orientar as equipes de engenheiros de software conforme realizam o trabalho de desenvolvimento de um sistema. Infelizmente, por vezes, tais objetivos não são alcançados. Se os modelos prescritivos forem aplicados de forma dogmática e sem adaptações, poderão aumentar a burocracia associada ao desenvolvimento de sistemas computacionais e, inadvertidamente, criarão dificuldades para todos os envolvidos.

Os *modelos ágeis de processo* (Capítulo 3) ressaltam a “agilidade” do projeto, seguindo princípios que conduzam a uma abordagem mais informal (porém, não menos eficiente) para o processo de software. Tais modelos de processo geralmente são caracterizados como “ágeis”, porque enfatizam a flexibilidade e adaptabilidade. Eles são apropriados para vários tipos de projetos e são particularmente úteis quando aplicações para a Internet são projetadas.

? Como os modelos de processo se diferenciam?

“Sinto que uma receita consiste em apenas um tema com o qual um cozinheiro inteligente pode brincar, cada vez, com uma variação.”

Madame Benoit

? O que caracteriza um processo “ágil”?

1.5 A PRÁTICA DA ENGENHARIA DE SOFTWARE

WebRef

Uma variedade de citações provocativas acerca da prática da engenharia de software pode ser encontrada em www.literateprogramming.com

A Seção 1.4 apresentou uma introdução a um modelo de processo de software genérico composto por um conjunto de atividades que estabelecem uma metodologia para a prática da engenharia de software. As atividades genéricas da metodologia — **comunicação, planejamento, modelagem, construção e emprego** —, bem como as atividades de apoio, estabelecem um esquema para o trabalho da engenharia de software. Mas como a prática da engenharia de software se encaixa nisso? Nas seções seguintes, você adquirirá um conhecimento básico dos princípios e conceitos genéricos que se aplicam às atividades de uma metodologia.¹²

1.5.1 A essência da prática

Em um livro clássico, *How to Solve It*, sobre os modernos computadores, George Polya [Pol45] apontou em linhas gerais a essência da solução de problemas e, conseqüentemente, a essência da prática da engenharia de software:

1. *Compreender o problema* (comunicação e análise).
2. *Planejar uma solução* (modelagem e projeto de software).

¹² Você deve rever seções relevantes contidas neste capítulo à medida que métodos de engenharia de software e atividades de apoio específicas forem discutidos posteriormente neste livro.



Pode-se afirmar que a abordagem de Polya é simplesmente questão de bom senso. É verdade. Mas é espantoso quão frequentemente o bom senso é incomum no mundo do software.

“Há um grão de descoberta na solução de qualquer problema.”

George Polya

3. *Executar o plano* (geração de código).

4. *Examinar o resultado para ter precisão* (testes e garantia da qualidade).

No contexto da engenharia de software, essas etapas de bom senso conduzem a uma série de questões essenciais [adaptado de Pol45]:

Compreenda o problema. Algumas vezes é difícil de admitir, porém, a maioria de nós é arrogante quando nos é apresentado um problema. Ouvimos por alguns segundos e então pensamos: “Ah, sim, estou entendendo, vamos começar a resolver este problema”. Infelizmente, compreender nem sempre é assim tão fácil. Vale a pena despende um pouco de tempo respondendo a algumas questões simples:

- *Quem tem interesse na solução do problema?* Ou seja, quem são os interessados?
- *Quais são as incógnitas?* Que dados, funções e recursos são necessários para resolver apropriadamente o problema?
- *O problema pode ser compartimentalizado?* É possível representá-lo em problemas menores que talvez sejam mais fáceis de ser compreendidos?
- *O problema pode ser representado graficamente?* É possível criar um modelo analítico?

Planeje a solução. Agora você entende o problema (ou assim pensa) e não vê a hora de começar a codificar. Antes de fazer isso, relaxe um pouco e faça um pequeno projeto:

- *Você já viu problemas similares anteriormente?* Existem padrões que são reconhecíveis em uma potencial solução? Existe algum software que implemente os dados, as funções e características necessárias?
- *Algum problema similar já foi resolvido?* Em caso positivo, existem elementos da solução que podem ser reutilizados?
- *É possível definir subproblemas?* Em caso positivo, existem soluções aparentes e imediatas para eles?
- *É possível representar uma solução de maneira que conduza a uma implementação efetiva?* É possível criar um modelo de projeto?

Execute/leve adiante o plano. O projeto elaborado que criamos serve como um mapa para o sistema que se quer construir. Podem surgir desvios inesperados e é possível que se descubra um caminho ainda melhor à medida que se prossiga, porém, o “planejamento” nos permitirá que continuemos sem nos perder.

- *A solução se adequa ao plano?* O código-fonte pode ser atribuído ao modelo de projeto?
- *Cada uma das partes componentes da solução está provavelmente correta?* O projeto e o código foram revistos, ou, melhor ainda, provas da correção foram aplicadas ao algoritmo?

Examine o resultado. Não se pode ter certeza de que uma solução seja perfeita, porém, pode-se assegurar que um número de testes suficiente tenha sido realizado para revelar o maior número de erros possível.

- *É possível testar cada parte componente da solução?* Foi implementada uma estratégia de testes razoável?
- *A solução produz resultados que se adequam aos dados, às funções e características necessários?* O software foi validado em relação a todas as solicitações dos interessados?

Não é surpresa que grande parte dessa metodologia consiste no bom senso. De fato, é razoável afirmar que uma abordagem de bom senso à engenharia de software jamais o levará ao mau caminho.



Antes de iniciar um projeto, certifique-se de que o software tem um propósito para a empresa e de que seus usuários reconheçam seu valor.

1.5.2 Princípios gerais

O dicionário define a palavra *princípio* como “uma importante afirmação ou lei subjacente em um sistema de pensamento”. Ao longo deste livro serão discutidos princípios em vários níveis de abstração. Alguns se concentram na engenharia de software como um todo, outros consideram uma atividade de metodologia genérica específica (por exemplo, **comunicação**) e outros ainda destacam as ações de engenharia de software (por exemplo, projeto de arquitetura) ou tarefas técnicas (por exemplo, redigir um cenário de uso). Independentemente do seu nível de enfoque, os princípios ajudam a estabelecer um modo de pensar para a prática segura da engenharia de software — esta é a razão porque são importantes.

David Hooker [Hoo96] propôs sete princípios que se concentram na prática da engenharia de software como um todo. Eles são reproduzidos nos parágrafos a seguir.¹³

Primeiro princípio: a razão de existir

Um sistema de software existe por uma única razão: *gerar valor a seus usuários*. Todas as decisões deveriam ser tomadas tendo esse princípio em mente. Antes de especificar uma necessidade de um sistema, antes de indicar alguma parte da funcionalidade de um sistema, antes de determinar as plataformas de hardware ou os processos de desenvolvimento, pergunte a si mesmo: “Isso realmente agrega valor real ao sistema?”. Se a resposta for “não”, não o faça. Todos os demais princípios se apoiam neste primeiro.

Segundo princípio: KISS (Keep It Simple, Stupid!, ou seja: Faça de forma simples, tapado!)

O projeto de software não é um processo casual; há muitos fatores a ser considerados em qualquer esforço de projeto — *todo projeto deve ser o mais simples possível, mas não tão simples assim*. Esse princípio contribui para um sistema mais fácil de compreender e manter. Isso não significa que características, até mesmo as internas, devam ser descartadas em nome da simplicidade.

De fato, frequentemente os projetos mais elegantes são os mais simples, o que não significa “rápido e malfeito” — na realidade, simplificar exige muita análise e trabalho durante as iterações, sendo que o resultado será um software de fácil manutenção e menos propenso a erros.

Terceiro princípio: mantenha a visão

Uma visão clara é essencial para o sucesso. Sem ela, um projeto se torna ambíguo. Sem uma integridade conceitual, corre-se o risco de transformar o projeto numa colcha de retalhos de projetos incompatíveis, unidos por parafusos inadequados... Comprometer a visão arquitetônica de um sistema de software debilita e até poderá destruir sistemas bem projetados. Ter um arquiteto responsável e capaz de manter a visão clara e de reforçar a adequação ajuda a assegurar o êxito de um projeto.

Quarto princípio: o que um produz outros consomem

Raramente um sistema de software de força industrial é construído e utilizado de forma isolada. De uma maneira ou de outra, alguém mais irá usar, manter, documentar ou, de alguma forma, dependerá da capacidade de entender seu sistema. Portanto, *sempre especifique, projete e implemente ciente de que alguém mais terá de entender o que você está fazendo*. O público para qualquer produto de desenvolvimento de software é potencialmente grande. Especifique tendo em vista os usuários; projete, tendo em mente os implementadores; e codifique considerando aqueles que terão de manter e estender o sistema. Alguém terá de depurar o código que você escreveu e isso o faz um usuário de seu código; facilitando o trabalho de todas essas pessoas você agrega maior valor ao sistema.

“Há uma certa majestade na simplicidade que está muito acima de toda a excentricidade do saber.”

Papa Alexandre (1688 - 1744)

PONTO-CHAVE

Um software de valor mudará ao longo de sua vida. Por essa razão, um software deve ser desenvolvido para fácil manutenção.

¹³ Reproduzido com a permissão do autor [Hoo96]. Hooker define padrões para esses princípios em <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>.

Quinto princípio: esteja aberto para o futuro

Um sistema com tempo de vida mais longo tem mais valor. Nos ambientes computacionais de hoje, em que as especificações mudam de um instante para outro e as plataformas de hardware se tornam rapidamente obsoletas, a vida de um software, em geral, é medida em meses. Entretanto, verdadeiros sistemas de software com força industrial devem durar um período muito maior — e, para isso, devem estar preparados para se adaptar a mudanças. Sistemas que obtêm sucesso são aqueles que foram projetados dessa forma desde seu princípio.

*Jamais faça projetos limitados, sempre pergunte “e se” e prepare-se para todas as possíveis respostas, criando sistemas que resolvam o problema geral, não apenas aquele específico.*¹⁴ Isso muito provavelmente conduziria à reutilização de um sistema inteiro.

Sexto princípio: planeje com antecedência, visando a reutilização

A reutilização economiza tempo e esforço;¹⁵ alcançar um alto grau de reuso é indiscutivelmente a meta mais difícil de ser atingida ao se desenvolver um sistema de software. A reutilização de código e projetos tem sido proclamada como o maior benefício do uso de tecnologias orientadas a objetos, entretanto, o retorno desse investimento não é automático. Alavancar as possibilidades de reutilização (oferecida pela programação orientada a objetos [ou convencional]) requer planejamento e capacidade de fazer previsões. Existem várias técnicas para levar a cabo a reutilização em cada um dos níveis do processo de desenvolvimento do sistema. *Planejar com antecedência para o reuso reduz o custo e aumenta o valor tanto dos componentes reutilizáveis quanto dos sistemas aos quais eles serão incorporados.*

Sétimo princípio: pense!

Este último princípio é, provavelmente, aquele que é mais menosprezado. *Pensar bem e de forma clara antes de agir quase sempre produz melhores resultados.* Quando se analisa alguma coisa, provavelmente esta sairá correta. Ganha-se também conhecimento de como fazer correto novamente. Se você realmente analisar algo e mesmo assim o fizer da forma errada, isso se tornará uma valiosa experiência. Um efeito colateral da análise é aprender a reconhecer quando não se sabe algo, e até que ponto poderá buscar o conhecimento. Quando a análise clara fez parte de um sistema, seu valor aflora. Aplicar os seis primeiros princípios requer intensa reflexão, para a qual as recompensas potenciais são enormes.

Se todo engenheiro de software e toda a equipe de software simplesmente seguissem os sete princípios de Hooker, muitas das dificuldades enfrentadas no desenvolvimento de complexos sistemas baseados em computador seriam eliminadas.

1.6 MITOS RELATIVOS AO SOFTWARE

“Na falta de padrões significativos substituindo o folclore, surge uma nova indústria como a do software.”

Tom DeMarco

Os mitos criados em relação ao software — crenças infundadas sobre o software e sobre o processo usado para criá-lo — remontam aos primórdios da computação. Os mitos possuem uma série de atributos que os tornam insidiosos. Por exemplo, eles parecem ser, de fato, afirmações razoáveis (algumas vezes contendo elementos de verdade), têm uma sensação intuitiva e frequentemente são promulgados por praticantes experientes “que entendem do riscado”.

Atualmente, a maioria dos profissionais versados na engenharia de software reconhece os mitos por aquilo que eles representam — atitudes enganosas que provocaram sérios problemas tanto para gerentes quanto para praticantes da área. Entretanto, antigos hábitos e atitudes são difíceis de ser modificados e resquícios de mitos de software permanecem.

¹⁴ Esse conselho pode ser perigoso se levado a extremos. Projetar para o “problema geral” algumas vezes requer compromissos de desempenho e pode tornar ineficientes as soluções específicas.

¹⁵ Embora isso seja verdade para aqueles que reutilizam o software em futuros projetos, a reutilização pode ser cara para aqueles que precisam projetar e desenvolver componentes reutilizáveis. Estudos indicam que o projeto e o desenvolvimento de componentes reutilizáveis pode custar de 25 a 200% mais que o próprio software. Em alguns casos, o diferencial de custo não pode ser justificado.

WebRef

Software Project Managers Network, no endereço www.spmn.com, pode ajudá-lo a dissipar esses e outros mitos.

Mitos de gerenciamento. Gerentes com responsabilidade sobre software, assim como gerentes da maioria das áreas, frequentemente estão sob pressão para manter os orçamentos, evitar deslizes nos cronogramas e elevar a qualidade. Como uma pessoa que está se afogando e se agarra a uma tábua, um gerente de software muitas vezes se agarra à crença num mito do software, para aliviar a pressão (mesmo que temporariamente).

Mito: Já temos um livro que está cheio de padrões e procedimentos para desenvolver software. Ele não supre meu pessoal com tudo que eles precisam saber?

Realidade: O livro com padrões pode muito bem existir, mas ele é usado? Os praticantes da área estão cientes de que ele existe? Esse livro reflete a prática moderna da engenharia de software? É completo? É adaptável? Está alinhado para melhorar o tempo de entrega, mantendo ainda o foco na qualidade? Em muitos casos, a resposta para todas essas perguntas é “não”.

Mito: Se o cronograma atrasar, poderemos acrescentar mais programadores e ficarmos em dia (algumas vezes denominado conceito da “horda mongol”).

Realidade: O desenvolvimento de software não é um processo mecânico como o de fábrica. Nas palavras de Brooks [Bro95]: “acrescentar pessoas num projeto de software atrasado só o tornará mais atrasado ainda”. A princípio, essa afirmação pode parecer um contrassenso, no entanto, o que ocorre é que, quando novas pessoas entram, as que já estavam terão de gastar tempo situando os recém-chegados, reduzindo, conseqüentemente, o tempo destinado ao desenvolvimento produtivo. Pode-se adicionar pessoas, mas somente de forma planejada e bem coordenada.

Mito: Se eu decidir terceirizar o projeto de software, posso simplesmente relaxar e deixar essa empresa realizá-lo.

Realidade: Se uma organização não souber gerenciar e controlar projetos de software, ela irá, invariavelmente, enfrentar dificuldades ao terceirizá-los.

Mitos dos clientes. O cliente solicitante do software computacional pode ser uma pessoa na mesa ao lado, um grupo técnico do andar de baixo, de um departamento de marketing/vendas, ou uma empresa externa que encomendou o projeto por contrato. Em muitos casos, o cliente acredita em mitos sobre software porque gerentes e profissionais da área pouco fazem para corrigir falsas informações. Mitos conduzem a falsas expectativas (do cliente) e, em última instância, à insatisfação com o desenvolvedor.

Mito: Uma definição geral dos objetivos é suficiente para começar a escrever os programas — podemos preencher detalhes posteriormente.

Realidade: Embora nem sempre seja possível uma definição ampla e estável dos requisitos, uma definição de objetivos ambígua é receita para um desastre. Requisitos não ambíguos (normalmente derivados da iteratividade) são obtidos somente pela comunicação contínua e eficaz entre cliente e desenvolvedor.

Mito: Os requisitos de software mudam continuamente, mas as mudanças podem ser facilmente assimiladas, pois o software é flexível.

Realidade: É verdade que os requisitos de software mudam, mas o impacto da mudança varia dependendo do momento em que ela foi introduzida. Quando as mudanças dos requisitos são solicitadas cedo (antes do projeto ou da codificação terem começado), o impacto sobre os custos é relativamente pequeno. Entretanto, conforme o tempo passa, ele aumenta rapidamente — recursos foram comprometidos, uma estrutura de projeto foi estabelecida e mudar pode causar uma revolução que exija recursos adicionais e modificações fundamentais no projeto.



Esforce-se ao máximo para compreender o que deve fazer antes de começar. Você pode não chegar a todos os detalhes, mas quanto mais você souber, menor será o risco.



Toda vez que pensar: “não temos tempo para engenharia de software”, pergunte a si mesmo, “teremos tempo para fazer de novo?”.

Mitos dos profissionais da área. Mitos que ainda sobrevivem nos profissionais da área têm resistido por mais de 50 anos de cultura de programação. Durante seus primórdios, a programação era vista como uma forma de arte. Modos e atitudes antigos dificilmente morrem.

- Mito:** Uma vez feito um programa e o colocado em uso, nosso trabalho está terminado.
- Realidade:** Uma vez alguém já disse que “o quanto antes se começar a codificar, mais tempo levará para terminá-lo”. Levantamentos indicam que entre 60 e 80% de todo o esforço será despendido após a entrega do software ao cliente pela primeira vez.
- Mito:** Até que o programa entre em funcionamento, não há maneira de avaliar sua qualidade.
- Realidade:** Um dos mecanismos de garantia da qualidade de software mais eficaz pode ser aplicado desde a concepção de um projeto — a revisão técnica. Revisores de software (descritos no Capítulo 15) são um “filtro de qualidade” que mostram ser mais eficientes do que testes para encontrar certas classes de defeitos de software.
- Mito:** O único produto passível de entrega é o programa em funcionamento.
- Realidade:** Um programa funcionando é somente uma parte de uma configuração de software que inclui muitos elementos. Uma variedade de produtos derivados (por exemplo, modelos, documentos, planos) constitui uma base para uma engenharia bem-sucedida e, mais importante, uma orientação para suporte de software.
- Mito:** A engenharia de software nos fará criar documentação volumosa e desnecessária e, invariavelmente, irá nos retardar.
- Realidade:** A engenharia de software não trata de criação de documentos, trata da criação de um produto de qualidade. Melhor qualidade conduz à redução do retrabalho, e menos retrabalho resulta em maior rapidez na entrega.

Muitos profissionais de software reconhecem a falácia dos mitos que acabamos de descrever. Lamentavelmente, métodos e atitudes habituais fomentam tanto gerenciamento quanto medidas técnicas deficientes, mesmo quando a realidade exige uma abordagem melhor. Ter ciência das realidades do software é o primeiro passo para buscar soluções práticas na engenharia de software.

1.7 Como Tudo Começou

Todo projeto de software é motivado por alguma necessidade de negócios — a necessidade de corrigir um defeito em uma aplicação existente; a necessidade de adaptar um “sistema legado” a um ambiente de negócios em constante transformação; a necessidade de estender as funções e os recursos de uma aplicação existente, ou a necessidade de criar um novo produto, serviço ou sistema.

No início de um projeto de software, a necessidade do negócio é, com frequência, expressa informalmente como parte de uma simples conversa. A conversa apresentada no quadro a seguir é típica.

Exceto por uma rápida referência, o software mal foi mencionado como parte da conversação. Ainda assim, o software irá decretar o sucesso ou o fracasso da linha de produtos *CasaSegura*. A empreitada de engenharia terá êxito apenas se o software para a linha *CasaSegura* tiver êxito; e o mercado irá aceitar o produto apenas se o software incorporado atender adequadamente às necessidades (ainda não declaradas) do cliente. Acompanharemos a evolução da engenharia do software *CasaSegura* em vários dos capítulos que estão por vir.

CASA SEGURA¹⁶**Como começa um projeto**

Cena: Sala de reuniões da CPI Corporation, empresa (fictícia) que fabrica produtos de consumo para uso doméstico e comercial.

Atores: Mal Golden, gerente sênior, desenvolvimento do produto; Lisa Perez, gerente de marketing; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo, desenvolvimento de negócios.

Conversa:

Joe: Lee, ouvi dizer que o seu pessoal está construindo algo. Do que se trata? Um tipo de caixa sem fio de uso amplo e genérico?

Lee: Trata-se de algo bem legal... Aproximadamente do tamanho de uma caixa de fósforos, conectável a todo tipo de sensor, como uma câmera digital — ou seja, é conectável a quase tudo. Usa o protocolo sem fio 802.11g, permitindo que acessemos saídas de dispositivos sem o emprego de fios. Acreditamos que nos levará a uma geração de produtos inteiramente nova.

Joe: Você concorda, Mal?

Mal: Sim. Na verdade, com as vendas tão em baixa quanto neste ano, precisamos de algo novo. Lisa e eu fizemos uma pequena pesquisa de mercado e acreditamos que conseguimos uma linha de produtos que poderá ser ampla.

Joe: Ampla em que sentido?

Mal (evitando comprometimento direto): Conte a ele sobre nossa ideia, Lisa.

Lisa: Trata-se de uma geração completamente nova na linha de “produtos de gerenciamento doméstico”. Chamamos esses produtos que criamos de CasaSegura. Eles usam uma nova interface sem fio e oferecem a pequenos empresários e proprietários de casas um sistema que é controlado por seus PCs, envolvendo segurança doméstica, sistemas de vigilância, controle de eletrodomésticos e dispositivos. Por exemplo, seria possível diminuir a temperatura do aparelho de ar condicionado enquanto você está voltando para casa, esse tipo de coisa.

Lee (reagindo sem pensar): O departamento de engenharia fez um estudo de viabilidade técnica dessa ideia, Joe. É possível fazê-lo com um baixo custo de fabricação. A maior parte dos componentes do hardware é encontrada no mercado; o software é um problema, mas não é nada que não possamos resolver.

Joe: Interessante, mas eu perguntei sobre o levantamento final.

Mal: Os PCs estão em mais de 70% dos lares, se formos capazes de estabelecer um preço baixo para essa coisa, ela poderia se tornar um produto “revolucionário”. Ninguém mais tem nosso dispositivo sem fio... Ele é exclusivo! Estaremos 2 anos à frente de nossos concorrentes... E as receitas? Algo em torno de 30 a 40 milhões de dólares no segundo ano...

Joe (sorrindo): Vamos levar isso adiante. Estou interessado.

1.8 RESUMO

Software é o elemento-chave na evolução de produtos e sistemas baseados em computador e uma das mais importantes tecnologias no cenário mundial. Ao longo dos últimos 50 anos, o software evoluiu de uma ferramenta especializada em análise de informações e resolução de problemas para uma indústria propriamente dita. Mesmo assim, ainda temos problemas para desenvolver software de boa qualidade dentro do prazo e orçamento estabelecidos.

Softwares — programas, dados e informações descritivas — contemplam uma ampla gama de áreas de aplicação e tecnologia. O software legado continua a representar desafios especiais àqueles que precisam fazer sua manutenção.

As aplicações e os sistemas baseados na Internet passaram de simples conjuntos de conteúdo informativo para sofisticados sistemas que apresentam funcionalidade complexa e conteúdo multimídia. Embora essas WebApps possuam características e requisitos exclusivos, elas não deixam de ser um tipo de software.

A engenharia de software engloba processos, métodos e ferramentas que possibilitam a construção de sistemas complexos baseados em computador dentro do prazo e com qualidade. O processo de software incorpora cinco atividades estruturais: comunicação, planejamento, modelagem, construção e emprego; e elas se aplicam a todos os projetos de software. A prática

¹⁶ O projeto *CasaSegura* será usado ao longo deste livro para ilustrar o funcionamento interno de uma equipe de projeto à medida que ela constrói um produto de software. A empresa, o projeto e as pessoas são inteiramente fictícias, porém as situações e os problemas são reais.

da engenharia de software é uma atividade de resolução de problemas que segue um conjunto de princípios básicos.

Inúmeros mitos em relação ao software continuam a levar gerentes e profissionais para o mau caminho, mesmo com o aumento do conhecimento coletivo de software e das tecnologias necessárias para construí-los. À medida que for aprendendo mais sobre a engenharia de software, você começará a compreender porque esses mitos devem ser derrubados toda vez que se deparar com eles.

PROBLEMAS E PONTOS A PONDERAR

- 1.1.** Cite pelo menos cinco outros exemplos de como a lei das consequências não intencionais se aplica ao software.
- 1.2.** Forneça uma série de exemplos (positivos e negativos) que indiquem o impacto do software em nossa sociedade.
- 1.3.** Desenvolva suas próprias respostas às cinco perguntas colocadas no início da Seção 1.1. Discuta-as com seus colegas.
- 1.4.** Muitas aplicações modernas mudam com frequência — antes de serem apresentadas ao usuário final e só então a primeira versão ser colocada em uso. Sugira algumas maneiras de construir software para impedir a deterioração decorrente de mudanças.
- 1.5.** Considere as sete categorias de software apresentadas na Seção 1.1.2. Você acha que a mesma abordagem em relação à engenharia de software pode ser aplicada a cada uma delas? Justifique sua resposta.
- 1.6.** A Figura 1.3 coloca as três camadas de engenharia de software acima de uma camada intitulada “foco na qualidade”. Isso implica um programa de qualidade organizacional como o de gestão da qualidade total. Pesquise um pouco a respeito e crie um sumário dos princípios básicos de um programa de gestão da qualidade total.
- 1.7.** A engenharia de software é aplicável quando as WebApps são construídas? Em caso positivo, como poderia ser modificada para atender às características únicas das WebApps?
- 1.8.** À medida que o software invade todos os setores, riscos ao público (devido a programas com imperfeições) passam a ser uma preocupação cada vez maior. Crie um cenário o mais catastrófico possível, porém realista, cuja falha de um programa de computador poderia causar um grande dano (em termos econômico ou humano).
- 1.9.** Descreva uma estrutura de processos com suas próprias palavras. Ao afirmarmos que atividades de modelagem se aplicam a todos os projetos, isso significa que as mesmas tarefas são aplicadas a todos os projetos, independentemente de seu tamanho e complexidade? Justifique.
- 1.10.** As atividades de apoio ocorrem ao longo do processo de software. Você acredita que elas são aplicadas de forma homogênea ao longo do processo ou algumas delas são concentradas em uma ou mais atividades de metodologia?
- 1.11.** Acrescente dois outros mitos à lista apresentada na Seção 1.6 e declare a realidade que acompanha tais mitos.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES¹⁷

Há literalmente milhares de livros sobre o tema software. A grande maioria trata de linguagens de programação ou aplicações de software, porém poucos tratam do software em si.

¹⁷ A seção *Leitura e fontes de informação adicionais* apresentada no final de cada capítulo apresenta uma breve visão geral de publicações que podem ajudar a expandir o seu entendimento dos principais tópicos apresentados no capítulo. Há um site bem abrangente para dar suporte ao livro *Engenharia de Software: uma abordagem prática* no endereço www.mhhe.com/pressman. Entre os diversos tópicos contemplados no site estão recursos de engenharia de software capítulo a capítulo e dicas de sites que poderão complementar o material apresentado em cada capítulo.

Pressman e Herron (*Software Shock*, Dorset House, 1991) apresentaram uma discussão preliminar (dirigida ao grande público) sobre software e a maneira pela qual os profissionais o desenvolvem. O best-seller de Negroponte (*Being Digital*, Alfred A. Knopf, Inc., 1995) dá uma visão geral da computação e seu impacto global no século XXI. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) produziu um conjunto de divertidos e perspicazes ensaios sobre software e o processo pelo qual ele é desenvolvido.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argumenta que o “flagelo moderno” dos bugs de software pode ser eliminado e sugere maneiras para concretizar isso. Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) defende que a “separação” entre aqueles que têm acesso a fontes de informação (por exemplo, a Web) e aqueles que não o têm está diminuindo, à medida que avançamos na primeira década deste século. Livros de Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) e Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introduzem o conceito de software “aberto” e preveem um ambiente sem fio no qual o software deve se adaptar às exigências que surgem em tempo real.

O estado atual da engenharia de software e do processo de software pode ser mais bem determinado a partir de publicações como *IEEE Software*, *IEEE Computer*, *CrossTalk* e *IEEE Transactions on Software Engineering*. Periódicos do setor como *Application Development Trends* e *Cutter IT Journal* normalmente contêm artigos sobre tópicos da engenharia de software. A disciplina é “sintetizada” todos os anos no *Proceeding of the International Conference on Software Engineering*, patrocinado pelo IEEE e ACM e é discutida de forma aprofundada em periódicos como *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes* e *Annals of Software Engineering*. Dezenas de milhares de sites são dedicados à engenharia de software e ao processo de software.

Foram publicados vários livros sobre o processo de desenvolvimento de software e sobre a engenharia de software nos últimos anos. Alguns fornecem uma visão geral de todo o processo, ao passo que outros se aprofundam em tópicos específicos importantes em detrimento dos demais. Entre as ofertas mais populares (além deste livro, é claro!), temos:

- Abran, A. e J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.
- Andersson, E. et al., *Software Engineering for Internet Applications*, The MIT Press, 2006.
- Christensen, M. e R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
- Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002.
- Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, 2.^a ed., Addison-Wesley, 2008.
- Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.
- Pfleeger, S., *Software Engineering: Theory and Practice*, 3.^a ed., Prentice-Hall, 2005.
- Schach, S., *Object-Oriented and Classical Software Engineering*, 7.^a ed., McGraw-Hill, 2006.
- Sommerville, I., *Software Engineering*, 8.^a ed., Addison-Wesley, 2006.
- Tsui, F. e O. Karam, *Essentials of Software Engineering*, Jones & Bartlett Publishers, 2006.

Foram publicados diversos padrões de engenharia de software pelo IEEE, pela ISO e suas organizações de padronização ao longo das últimas décadas. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) disponibiliza uma pesquisa útil de padrões relevantes e como aplicá-los a projetos reais.

Uma ampla gama de fontes de informação sobre engenharia de software e processo de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes para o processo para o software pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PROCESSOS DE SOFTWARE

Nesta seção você aprenderá sobre o processo que fornece uma metodologia para a prática da engenharia de software. As questões abaixo são tratadas nos capítulos seguintes:

- O que é um processo de software?
- Quais são as atividades metodológicas genéricas presentes em todos os processos de software?
- Como os processos são modelados e o que são padrões de processo?
- O que são modelos de processo prescritivo e quais são seus pontos fortes e fracos?
- Por que agilidade é um lema no trabalho da engenharia de software moderna?
- O que é desenvolvimento de software ágil e como ele difere dos modelos de processos mais tradicionais?

Respondidas tais questões, você estará mais bem preparado para compreender o contexto no qual a prática da engenharia de software é aplicada.

2

MODELOS DE PROCESSO

CONCEITOS- -CHAVE

conjunto de tarefas	55
desenvolvimento baseado em componentes.	69
modelo de métodos formais.	69
modelo de processo genérico	53
modelos concorrentes.	67
modelos de processo evolucionário	62
modelos de processo incremental.	61
modelos de processo prescritivo	58
padrões de processo.	55
processo de software em equipe.	75
processo de software pessoal.	74
processo unificado.	71

Em um livro fascinante, que nos dá a visão de um economista acerca do software e da engenharia de software, Howard Baetjer, Jr. [Bae98], comenta o processo de software:

Pelo fato de software, como todo capital, ser conhecimento incorporado, e pelo fato de esse conhecimento ser, inicialmente, disperso, tácito, latente e em considerável medida, incompleto, o desenvolvimento de software é um processo de aprendizado social. Esse processo é um diálogo no qual o conhecimento, que deverá tornar-se o software, é coletado, reunido e incorporado ao software. Tal processo possibilita a interação entre usuários e projetistas, entre usuários e ferramentas em evolução e entre projetistas e ferramentas em evolução (tecnologia). Trata-se de um processo iterativo no qual a própria ferramenta em evolução serve como meio de comunicação, com cada nova rodada do diálogo extraindo mais conhecimento útil das pessoas envolvidas.

De fato, construir software é um processo de aprendizado social iterativo e o resultado, algo que Baetjer denominaria “capital de software”, é a incorporação do conhecimento coletado, filtrado e organizado conforme se desenvolve o processo.

Mas o que é exatamente um processo de software do ponto de vista técnico? No contexto desse livro, *processo de software é definido* como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver um software de alta qualidade. “Processo” é sinônimo de engenharia de software? A resposta é “sim e não”. Um processo de software define a abordagem adotada conforme um software é elaborado pela engenharia. Mas a engenharia de software também engloba tecnologias que fazem parte do processo — métodos técnicos e ferramentas automatizadas.

Mais importante, a engenharia de software é realizada por pessoas criativas e com amplos conhecimentos e que devem adaptar um processo de software maduro, de forma que fique apropriado aos produtos desenvolvidos e às demandas de seu mercado.

PANORAMA

O que é? Quando se trabalha na elaboração de um produto ou sistema, é importante seguir uma série de passos previsíveis — um roteiro que ajude a criar um resultado de alta qualidade e dentro do prazo estabelecido. O roteiro é denominado “processo de software”.

Quem realiza? Os engenheiros de software e seus gerentes adaptam o processo às suas necessidades e então o seguem. Os solicitantes do software têm um papel a desempenhar no processo de definição, construção e teste do software.

Por que ele é importante? Porque propicia estabilidade, controle e organização para uma atividade que pode, sem controle, tornar-se bastante caótica. Entretanto, uma abordagem de engenharia de software moderna deve ser “ágil”. Deve demandar apenas atividades, controles e produtos de trabalho que sejam apropriados para a equipe do projeto e para o produto a ser produzido.

Quais são as etapas envolvidas? O processo adotado depende do software a ser desenvolvido. Um determinado processo pode ser apropriado para um software do sistema “aviônico” de uma aeronave, enquanto um processo totalmente diferente pode ser indicado para a criação de um site.

Qual é o artefato? Do ponto de vista de um engenheiro de software, os produtos de trabalho são os programas, os documentos e os dados produzidos em consequência das atividades e tarefas definidas pelo processo.

Como garantir que o trabalho foi feito corretamente? Há muitos mecanismos de avaliação dos processos de software que possibilitam às organizações determinarem o nível de “maturidade” de seu processo de software. Entretanto, a qualidade, o cumprimento de prazos e a viabilidade a longo prazo do produto que se desenvolve são os melhores indicadores da eficácia do processo utilizado.

2.1 UM MODELO DE PROCESSO GENÉRICO

No Capítulo 1, processo foi definido como um conjunto de atividades de trabalho, ações e tarefas realizadas quando algum artefato de software deve ser criado. Cada uma dessas atividades, ações e tarefas alocam-se dentro de uma metodologia ou modelo que determina seu relacionamento com o processo e seu relacionamento umas com as outras.

O processo de software é representado esquematicamente na Figura 2.1. De acordo com a figura, cada atividade metodológica é composta por um conjunto de ações de engenharia de software. Cada ação é definida por um *conjunto de tarefas*, o qual identifica as tarefas de trabalho a ser completadas, os artefatos de software que serão produzidos, os fatores de garantia da qualidade que serão exigidos e os marcos utilizados para indicar progresso.

Como discutido no Capítulo 1, uma metodologia de processo genérica para engenharia de software estabelece cinco atividades metodológicas: **comunicação, planejamento, modelagem, construção e entrega**. Além disso, um conjunto de atividades de apoio (*umbrella activities*) são aplicadas ao longo do processo, como o acompanhamento e controle do projeto, a administração de riscos, a garantia da qualidade, o gerenciamento das configurações, as revisões técnicas e outras.

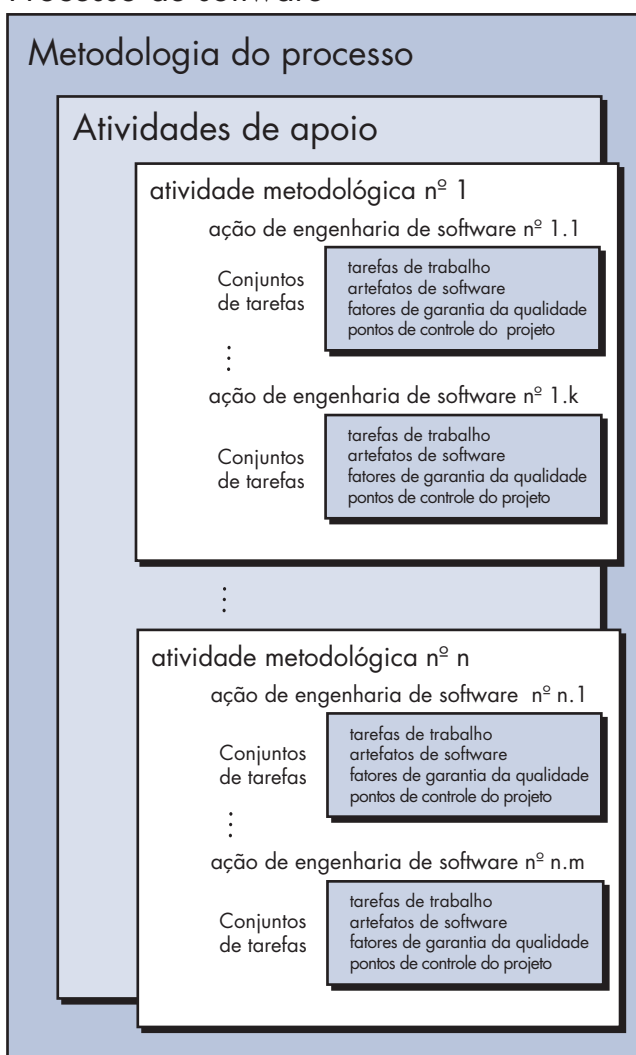
PONTO-CHAVE

A hierarquia de trabalho técnico, dentro do processo de software, consiste em: atividades, ações abrangentes, compostas por tarefas.

FIGURA 2.1

Uma metodologia do processo de software

Processo de software



"Achamos que desenvolvedores de software não percebem uma verdade essencial: a maioria das organizações não sabe o que faz. Elas acham que sabem, mas não sabem."

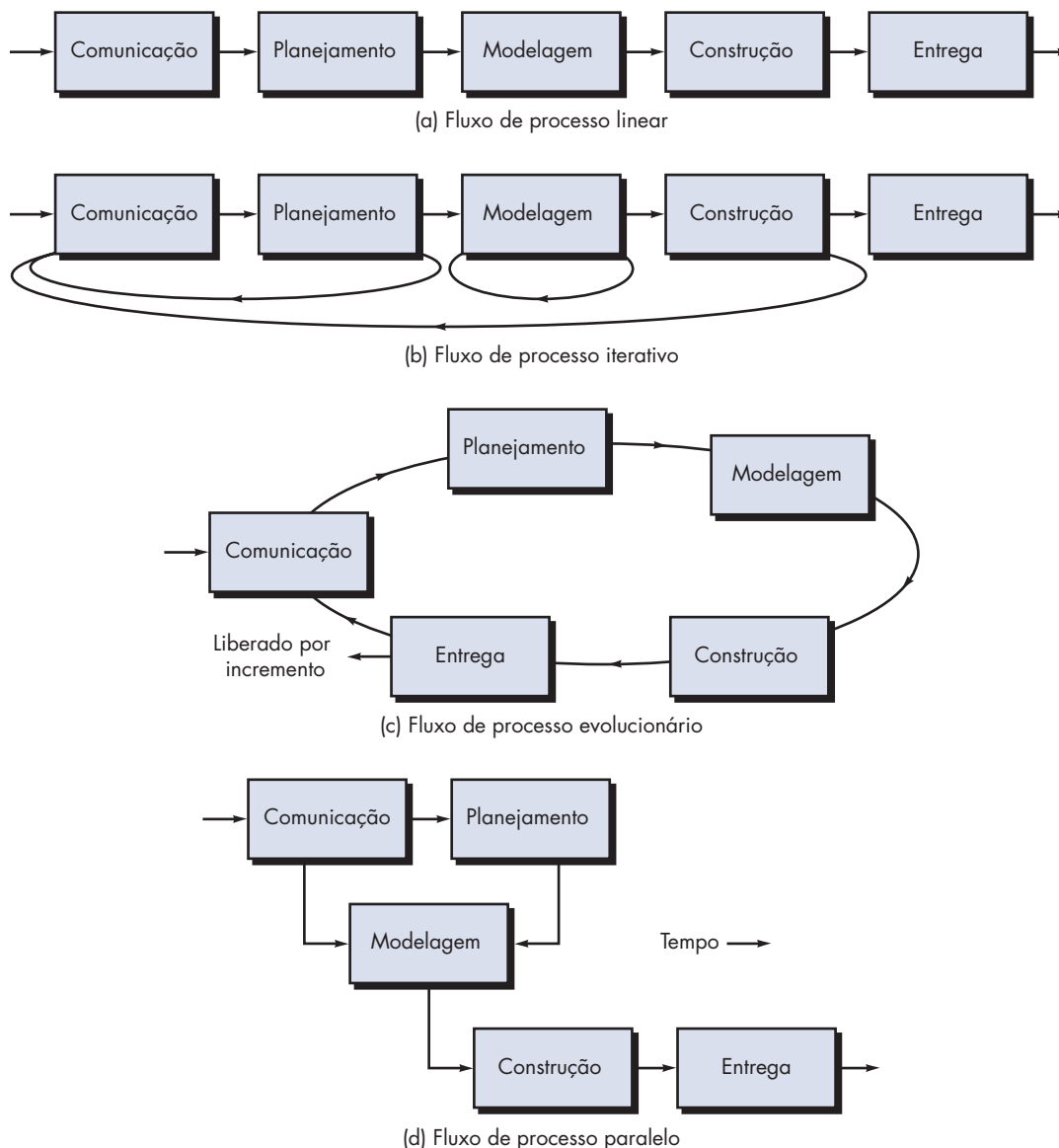
Tom DeMarco

Deve-se notar que um importante aspecto do processo de software ainda não foi discutido. Esse aspecto — chamado *fluxo de processo* — descreve como são organizadas as atividades metodológicas, bem como as ações e tarefas que ocorrem dentro de cada atividade em relação à sequência e ao tempo, como ilustrado na Figura 2.2.

Um *fluxo de processo linear* executa cada uma das cinco atividades metodológicas em sequência, começando com a de comunicação e culminando com a do emprego (Figura 2.2a). Um *fluxo de processo iterativo* repete uma ou mais das atividades antes de prosseguir para a seguinte (Figura 2.2b). Um *fluxo de processo evolucionário* executa as atividades de uma forma "circular". Cada volta pelas cinco atividades conduz a uma versão mais completa do software (Figura 2.2c). Um *fluxo de processo paralelo* (Figura 2.2d) executa uma ou mais atividades em paralelo com outras atividades (por exemplo, a modelagem para um aspecto do software poderia ser executada em paralelo com a construção de um outro aspecto do software).

FIGURA 2.2

Fluxo de processo



? Como uma atividade metodológica é modificada de acordo com as alterações da natureza do projeto?

2.1.1 Definindo atividade metodológica

Embora tenham sido descritas cinco atividades metodológicas e se tenha fornecido uma definição básica de cada uma delas no Capítulo 1, uma equipe de software precisa de muito mais informações antes de poder executar apropriadamente qualquer uma dessas atividades como parte do processo de software. Consequentemente, depara-se com uma questão-chave: *Que ações são apropriadas para uma atividade metodológica, uma vez fornecidos a natureza do problema a ser solucionado, as características das pessoas que estarão executando o trabalho e os interessados que estarão propondo o projeto?*

Para um pequeno projeto de software solicitado por uma única pessoa (numa localidade longínqua) com necessidades simples e objetivas, a atividade de comunicação pode resumir-se a pouco mais de um telefonema para o devido solicitante. Portanto, a única ação necessária é uma *conversação telefônica*, e as tarefas de trabalho (o *conjunto de tarefas*) que essa ação envolve são:

1. Contactar o interessado via telefone.
2. Discutir as necessidades e tomar notas.
3. Organizar anotações em uma breve relação de requisitos, por escrito.
4. Enviar um e-mail para o interessado para revisão e aprovação.

Se o projeto fosse consideravelmente mais complexo, com muitos interessados, cada qual com um conjunto de necessidades diferentes (por vezes conflitantes), a atividade de comunicação poderia ter seis ações distintas (descritas no Capítulo 5): *inserção*, *elucidação*, *elaboração*, *negociação*, *especificação* e *validação*. Cada uma dessas ações de engenharia de software conteria muitas tarefas de trabalho e uma série de diferentes artefatos.

PONTO-CHAVE

Projetos diferentes demandam conjuntos de tarefas diferentes. A equipe de software escolhe o conjunto de tarefas fundamentada no problema e nas características do projeto.

2.1.2 Identificação de um conjunto de tarefas

Referindo-se novamente à Figura 2.1, cada ação de engenharia de software (por exemplo, *elucidação*, uma ação associada à atividade de comunicação) pode ser representada por vários e diferentes *conjuntos de tarefas* — cada um constituído por uma gama de tarefas de trabalho de engenharia de software, artefatos relativos, fatores de garantia da qualidade e pontos de controle do projeto. Deve-se escolher um conjunto de tarefas mais adequado às necessidades do projeto e às características da equipe. Isso significa que uma ação de engenharia de software pode ser adaptada às necessidades específicas do projeto de software e às características da equipe.

2.1.3 Padrões de processos

Toda equipe de desenvolvimento encontra problemas à medida que avança no processo de software. Seria útil se soluções comprovadas estivessem prontamente à disposição da equipe, de modo que os problemas pudessem ser localizados e resolvidos rapidamente. Um *padrão de processo*¹ descreve um problema de processo encontrado durante o trabalho de engenharia de software, identificando o ambiente onde foi encontrado e sugerindo uma ou mais soluções comprovadas para o problema. Em termos mais genéricos, um padrão de processo fornece um modelo (template) [Amb98] — um método consistente para descrever soluções de problemas no contexto do processo de software. Combinando padrões, uma equipe conseguirá solucionar problemas e elaborar um processo que melhor atenda às necessidades de um projeto.

Padrões podem ser definidos em qualquer nível de abstração.² Em alguns casos, um padrão poderia ser utilizado para descrever um problema (e sua solução) associado ao modelo de processo completo (por exemplo, prototipação). Em outras situações, os padrões podem ser usados para descrever um problema (e sua solução) associado a uma atividade metodológica (por

? O que é padrão de processo?

“A repetição de padrões é algo bem diferente da repetição de partes. De fato, as partes diferentes serão únicas, pois os padrões são os mesmos.”

Christopher Alexander

¹ Uma discussão detalhada sobre padrões é apresentada no Capítulo 12.

² Os padrões são aplicáveis a várias atividades de engenharia de software. Padrões de análise, de projeto e de testes são discutidos nos Capítulos 7, 9, 10, 12 e 14. Padrões e “antipadrões” para atividades de gerenciamento de projetos são discutidos na Parte 4 deste livro.

INFORMAÇÕES



Conjunto de tarefas

Um conjunto de tarefas define o verdadeiro trabalho a ser feito para se atingir os objetivos de uma ação de engenharia de software. Por exemplo, elucidação (mais comumente denominada “levantamento de requisitos”) é uma importante ação de engenharia de software que ocorre durante a atividade de comunicação. A meta do levantamento de requisitos é compreender o que os vários interessados esperam do software a ser desenvolvido.

Para um projeto pequeno, relativamente simples, o conjunto de tarefas para levantamento das necessidades seria semelhante ao seguinte:

1. Fazer uma lista dos envolvidos no projeto.
2. Fazer uma reunião informal com todos os interessados.
3. Solicitar para cada interessado uma lista com as características e funções necessárias.
4. Discutir sobre os requisitos e construir uma lista final.
5. Organizar os requisitos por grau de prioridade.
6. Destacar pontos de incertezas.

Para um projeto de software maior e mais complexo, necessitaria-se de um conjunto diferente de tarefas. Tal conjunto pode incluir as seguintes tarefas de trabalho:

1. Fazer uma lista dos envolvidos no projeto.
2. Entrevistar separadamente cada um dos envolvidos para levantamento geral de suas expectativas e necessidades.

3. Fazer uma lista preliminar das funções e características, com base nas informações fornecidas pelos interessados.
4. Agendar uma série de reuniões facilitadoras para especificação de aplicações.
5. Promover reuniões.
6. Incluir cenários informais de usuários como parte de cada reunião.
7. Aprimorar os cenários de usuários, com base no *feedback* dos interessados.
8. Fazer uma lista revisada das necessidades dos interessados.
9. Empregar técnicas de aplicação de funções de qualidade para estabelecer graus de prioridade dos requisitos.
10. Agrupar os requisitos de modo que eles possam ser entregues incrementalmente.
11. Fazer um levantamento das limitações e restrições que serão aplicadas ao sistema.
12. Discutir sobre os métodos para validação do sistema.

Esses dois conjuntos de tarefas atingem o objetivo de “levantamento de necessidades”, porém, são bem diferentes em relação aos graus de profundidade e formalidade. A equipe de software deve escolher o conjunto de tarefas que lhe possibilitará atingir o objetivo de cada ação, mantendo, inclusive, a qualidade e a agilidade.

exemplo, **planejamento**) ou uma ação dentro de uma atividade metodológica (por exemplo, estimativa de custos do projeto).

Ambler [Amb98] propôs um modelo para descrever um padrão de processo:

Nome do Padrão. O padrão deve receber um nome significativo que o descreva no contexto do processo de software (por exemplo, **RevisõesTécnicas**).

Forças. Ambiente onde se encontram o padrão e as questões que tornam visível o problema e que poderiam afetar sua solução.

Tipo. É especificado o tipo de padrão. Ambler sugere três tipos:

1. *Padrão de estágio* — define um problema associado a uma atividade metodológica para o processo. Como uma atividade metodológica envolve múltiplas ações e tarefas de trabalho, um padrão de estágio engloba múltiplos padrões de tarefas (veja o próximo padrão) que são relevantes ao estágio (atividade metodológica). Podemos citar como um exemplo de padrão de estágio **EstabelecendoComunicação**. Esse padrão incorpora o padrão de tarefas **LevantamentodeNecessidades** e outros.
2. *Padrão de tarefas* — define um problema associado a uma ação de engenharia de software ou tarefa de trabalho relevante para a prática de engenharia de software bem-sucedida (por exemplo, **LevantamentodeNecessidades** é um padrão de tarefas).
3. *Padrão de fases* — define a sequência das atividades metodológicas que ocorrem dentro do processo, mesmo quando o fluxo geral de atividades é iterativo por natureza. Um exemplo de padrão de fases seria **ModeloEspiral** ou **Prototipação**.³

PONTO-CHAVE

Um modelo de padrões propicia um meio consistente para descrever um padrão.

3 Esses padrões de fases são discutidos na Seção 2.3.3.

Contexto Inicial. Descreve as condições sob as quais o padrão se aplica. Antes do início do padrão: (1) Que atividades organizacionais ou relacionadas à equipe já ocorreram? (2) Qual o estado inicial para o processo? (3) Que informação de engenharia de software ou de projeto já existe?

Por exemplo, o padrão **Planejamento** (um padrão de estágio) requer que: (1) clientes e engenheiros de software tenham estabelecido uma comunicação colaborativa; (2) Tenha ocorrido a finalização bem-sucedida de uma série de padrões de tarefas [especificados] para o padrão **Comunicação**; e (3) Sejam conhecidos o escopo do projeto, as necessidades básicas do negócio, bem como as restrições do projeto.

Problema. O problema específico a ser resolvido pelo padrão.

Solução. Descreve como implementar o padrão de forma bem-sucedida. Esta seção descreve como o estado inicial do processo (que existe antes de o padrão ser implementado) é modificado como consequência do início do padrão. Descreve também como as informações de engenharia de software ou de projeto que se encontram à disposição antes do início do padrão são transformadas como consequência da execução do padrão de forma bem-sucedida.

Contexto Resultante. Descreve as condições que resultarão assim que o padrão tiver sido implementado com êxito. Após a finalização do padrão:

- (1) Quais atividades organizacionais ou relacionadas à equipe devem ter ocorrido?
- (2) Qual é o estado de saída para o processo?
- (3) Quais informações de engenharia de software ou de projeto foram desenvolvidas?

Padrões Relativos. Fornece uma lista de todos os padrões de processo que estão diretamente relacionados ao processo em questão. Essa lista pode ser representada de forma hierárquica ou em alguma outra forma com diagramas. Por exemplo, o padrão de estágio **Comunicação** envolve os padrões de tarefas: **EquipeDeProjeto**, **DiretrizesColaborativas**, **IsolamentoDoEscopo**, **LevantamentoDeNecessidades**, **DescriçãoDasRestrições** e **CriaçãoDeCenários**.



Um exemplo de padrão de processo

O padrão de processo sintetizado a seguir descreve uma abordagem que pode ser aplicada quando os interessados têm uma ideia geral do que precisa ser feito, mas estão incertos quanto aos requisitos específicos do software.

Nome do padrão. RequisitosImprecisos

Intento. Este padrão descreve uma abordagem voltada para a construção de um modelo (um protótipo) passível de ser avaliado iterativamente pelos interessados, num esforço para identificar ou solidificar os requisitos de software.

Tipo. Padrão de fase.

Contexto inicial. As condições seguintes devem ser atendidas antes de iniciar esse padrão: (1) interessados identificados; (2) forma de comunicação entre interessados e equipe de software já determinada; (3) principal problema de software a ser resolvido já identificado pelos interessados; (4) compreensão inicial do escopo do projeto, dos requisitos de negócio básicos e das restrições do projeto já atingida.

Problema. Os requisitos são vagos ou inexistentes, ainda assim há um reconhecimento claro de que existe um problema a

ser solucionado e este deve ser identificado utilizando-se uma solução de software. Os interessados não sabem o que querem, ou seja, eles não conseguem descrever os requisitos de software em detalhe.

Solução. Uma descrição do processo de prototipação poderia ser apresentada nesta etapa, mas é descrita posteriormente na Seção 2.3.3.

Contexto resultante. Um protótipo de software que identifique os requisitos básicos (por exemplo, modos de interação, características computacionais, funções de processamento) é aprovado pelos interessados. Em seguida, (1) o protótipo pode evoluir por uma série de incrementos para se tornar o software de produção ou (2) o protótipo pode ser descartado e o software de produção ser construído usando-se algum outro padrão de processos.

Padrões associados. Os seguintes padrões estão relacionados a esse padrão: **ComunicaçãoComOCliente**, **ProjetoIterativo**, **DesenvolvimentoIterativo**, **AvaliaçãoDoCliente**, **ExtraçãoDeRequisitos**.

Usos conhecidos e um exemplo. A prototipação é recomendada quando as necessidades são incertas.

INFORMAÇÕES

Usos Conhecidos e Exemplos. Indicam as instâncias específicas onde o padrão é aplicável. Por exemplo, **Comunicação** é obrigatória no início de todo projeto de software, é recomendável ao longo de todo o projeto de software e é obrigatória assim que a atividade de emprego estiver em andamento.

WebRef

Recursos completos para padrões de processo podem ser encontrados em www.ambyssoft.com/processPatternsPage.html.

Padrões de processo propiciam um mecanismo efetivo para localização de problemas associados a qualquer processo de software. Os padrões permitem que se desenvolva uma descrição de processo de forma hierárquica que se inicia com nível alto de abstração (um padrão de fases). A descrição é então refinada em um conjunto de padrões de estágio que descreve atividades metodológicas e são ainda mais refinadas de uma forma hierárquica, em padrões de tarefa mais detalhados para cada padrão de estágio. Uma vez que os padrões de processos tenham sido desenvolvidos, eles poderão ser reutilizados para a definição de variantes de processo — isto é, um modelo de processo personalizado pode ser definido por uma equipe de software usando os padrões como blocos de construção para o modelo de processo.

2.2 AVALIAÇÃO E APERFEIÇOAMENTO DE PROCESSOS

PONTO-CHAVE

Tentativas de avaliação para compreender o atual estado do processo de software com o intuito de aperfeiçoá-lo.

A existência de um processo de software não garante que o software será entregue dentro do prazo, que estará de acordo com as necessidades do cliente ou que apresentará características técnicas que conduzirão a características de qualidade de longo prazo (Capítulos 14 e 16). Os padrões de processo devem ser combinados com uma prática de engenharia de software consistente (Parte 2 deste livro). Além disso, o próprio processo pode ser avaliado para assegurar que está de acordo com um conjunto de critérios de processo básicos comprovados como essenciais para uma engenharia de software de sucesso.⁴

Ao longo das últimas décadas foi proposta uma série de abordagens diferentes em relação à avaliação e ao aperfeiçoamento dos processos de software:

Quais técnicas formais estão disponíveis para avaliar o processo de software?

SCAMPI (Standard CMMI Assessment Method for Process Improvement) — (Método Padrão CMMI de Avaliação para Aperfeiçoamento de Processo da CMMI): fornece um modelo de avaliação do processo de cinco etapas, contendo cinco fases: início, diagnóstico, estabelecimento, atuação e aprendizado. O método SCAMPI usa o CMMI da SEI como base para avaliação [SEI00].

CBA IPI (CMM — Based Appraisal for Internal Process Improvement) — (Avaliação para Aperfeiçoamento do Processo Interno baseada na CMM): fornece uma técnica de diagnóstico para avaliar a maturidade relativa de uma organização de software; usa a CMM da SEI como base para a avaliação [Dun01].

SPICE (ISO/IEC15504) — padrão que define um conjunto de requisitos para avaliação do processo de software. A finalidade do padrão é auxiliar as organizações no desenvolvimento de uma avaliação objetiva da eficácia de um processo qualquer de software [ISO08].

ISO 9001:2000 para Software — padrão genérico aplicável a qualquer organização que queira aperfeiçoar a qualidade global de produtos, sistemas ou serviços fornecidos. Portanto, o padrão é aplicável diretamente a organizações e empresas de software [Ant06].

Uma discussão mais detalhada sobre métodos de avaliação de software e aperfeiçoamento de processo é apresentada no Capítulo 30.

“As organizações de software apresentaram falhas significativas quanto à habilidade em capitalizar as experiências adquiridas nos projetos finalizados.”

Nasa

2.3 MODELOS DE PROCESSO PRESCRITIVO

Originalmente, modelos de processo prescritivo foram propostos para trazer ordem ao caos existente na área de desenvolvimento de software. A história tem demonstrado que esses

⁴ A CMMI [CMM07] da SEI descreve, de forma extremamente detalhada, as características de um processo de software e os critérios para o êxito de um processo.

“Se o processo estiver correto, os resultados falarão por si mesmos.”

Takashi Osada

modelos tradicionais proporcionaram uma considerável contribuição quanto à estrutura utilizável no trabalho de engenharia de software e forneceram um roteiro razoavelmente eficaz para as equipes de software. Entretanto, o trabalho de engenharia de software e o seu produto permanecem “à beira do caos”.

Em um intrigante artigo sobre o estranho relacionamento entre ordem e caos no mundo do software, Nogueira e seus colegas [Nog00] afirmam que

O limiar do caos é definido como “um estado natural entre ordem e caos, um grande compromisso entre estrutura e surpresa” [Kau95]. O limiar do caos pode ser visualizado como um estado instável, parcialmente estruturado... Instável porque é constantemente atraído para o caos ou para a ordem absoluta.

Temos uma tendência de pensar que ordem é o estado ideal da natureza. Isso pode ser um erro. Pesquisas... Defendem a teoria de que a operação longe do equilíbrio gera criatividade, processos auto-organizados e lucros crescentes [Roo96]. Ordem absoluta implica ausência de variabilidade, o que poderia ser uma vantagem em ambientes imprevisíveis. A mudança ocorre quando existe uma estrutura que permita que a mudança possa ser organizada, mas tal estrutura não deve ser tão rígida a ponto de impedir que a mudança ocorra. Por outro lado, caos em demasia pode tornar impossível a coordenação e a coerência. A falta de estrutura nem sempre implica desordem.

As implicações filosóficas desse argumento são significativas para a engenharia de software. Se os modelos de processos prescritivos⁵ buscam ao máximo a estrutura e a ordem, seriam eles inapropriados para um mundo de software que prospera com as mudanças? E mais, se rejeitarmos os modelos de processo tradicionais (e a ordem implícita) e substituí-los por algo menos estruturado, tornaríamos impossível atingir a coordenação e a coerência no trabalho de software?

Não há respostas fáceis para essas questões, mas existem alternativas disponíveis para os engenheiros de software. Nas seções seguintes, examina-se a abordagem de processos prescritivos no qual a ordem e a consistência do projeto são questões dominantes. Denominam-se “prescritivos” porque prescrevem um conjunto de elementos de processo — atividades metodológicas, ações de engenharia de software, tarefas, produtos de trabalho, garantia da qualidade e mecanismos de controle de mudanças para cada projeto. Cada modelo de processo também prescreve um fluxo de processo (também denominado *fluxo de trabalho*) — ou seja, a forma pela qual os elementos do processo estão inter-relacionados.

Todos os modelos de processo de software podem acomodar as atividades metodológicas genéricas descritas no Capítulo 1, porém, cada um deles dá uma ênfase diferente a essas atividades e define um fluxo de processo que invoca cada atividade metodológica (bem como tarefas e ações de engenharia de software) de forma diversa.

2.3.1 O modelo cascata

há casos em que os requisitos de um problema são bem compreendidos — quando o trabalho flui da **comunicação** ao **emprego** de forma relativamente linear. Essa situação ocorre algumas vezes quando adaptações ou aperfeiçoamentos bem definidos precisam ser feitos em um sistema existente (por exemplo, uma adaptação em software contábil exigida devido a mudanças nas normas governamentais). Pode ocorrer também em um número limitado de novos esforços de desenvolvimento, mas apenas quando os requisitos estão bem definidos e são razoavelmente estáveis.

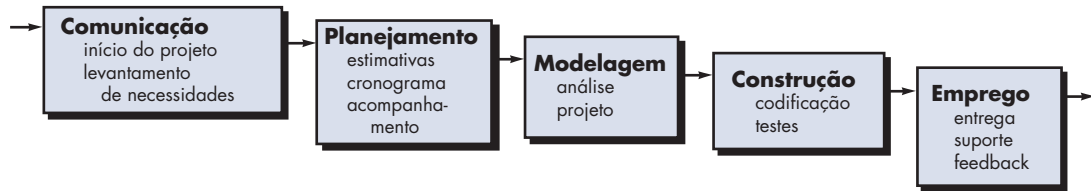
O *modelo cascata*, algumas vezes chamado *ciclo de vida clássico*, sugere uma abordagem sequencial e sistemática⁶ para o desenvolvimento de software, começando com o levantamento de necessidades por parte do cliente, avançando pelas fases de planejamento, modelagem, construção, emprego e culminando no suporte contínuo do software concluído (Figura 2.3).

⁵ Os modelos de processos prescritivos são, algumas vezes, conhecidos como modelos de processos “tradicionais”.

⁶ Embora o modelo cascata proposto por Winston Royce [Roy70] preveja os “feedback loops”, a vasta maioria das organizações que aplica esse modelo de processo os trata como se fossem estritamente lineares.

PONTO-CHAVE

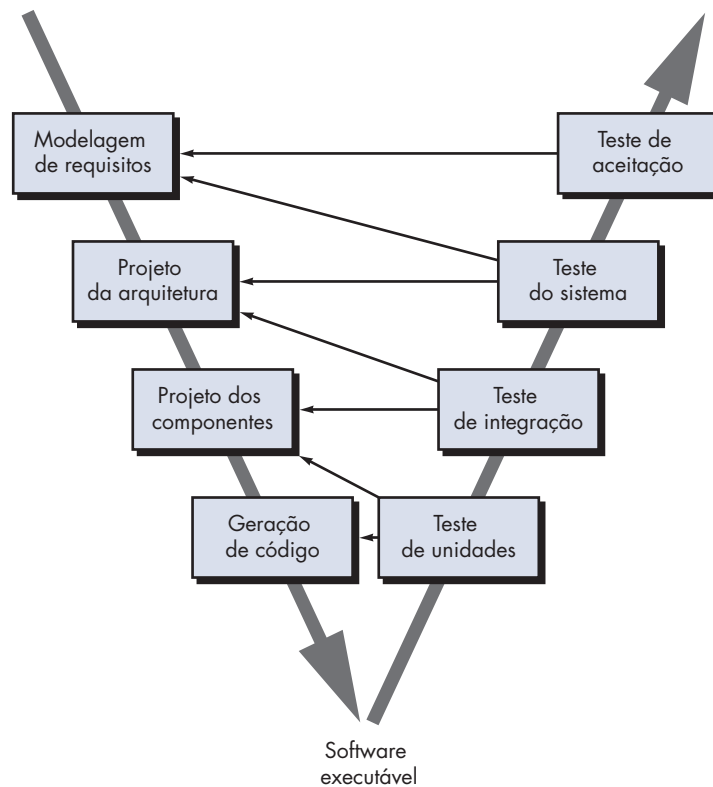
Os modelos de processo preceptivo definem um conjunto prescrito de elementos de processo e um fluxo de trabalho de processo previsível.

FIGURA 2.3**O modelo cascata****PONTO-CHAVE**

O modelo V ilustra como as ações de verificação e validação estão associadas a ações de engenharia anteriores.

Uma variação na representação do modelo cascata é denominada *modelo V*. Representado na Figura 2.4, o modelo V [Buc99] descreve a relação entre ações de garantia da qualidade e as ações associadas à comunicação, modelagem e atividades de construção iniciais. À medida que a equipe de software desce em direção ao lado esquerdo do V, os requisitos básicos do problema são refinados em representações progressivamente cada vez mais detalhadas e técnicas do problema e de sua solução. Uma vez que o código tenha sido gerado, a equipe se desloca para cima, no lado direito do V, realizando basicamente uma série de testes (ações de garantia da qualidade) que validem cada um dos modelos criados à medida que a equipe se desloca para baixo, no lado esquerdo do V.⁷ Na realidade, não existe uma diferença fundamental entre o ciclo de vida clássico e o modelo V. O modelo V fornece uma forma para visualizar como a verificação e as ações de validação são aplicadas ao trabalho de engenharia anterior.

O modelo cascata é o paradigma mais antigo da engenharia de software. Entretanto, ao longo das últimas três décadas, as críticas a este modelo de processo fez com que até mesmo seus mais ardentes defensores questionassem sua eficácia [Han95]. Entre os problemas às vezes encontrados quando se aplica o modelo cascata, temos:

FIGURA 2.4**Modelo V**

7 Na Parte 3 deste livro é apresentada uma discussão detalhada sobre ações de garantia da qualidade.

? Por que algumas vezes o modelo cascata falha?

“Muito frequentemente, o trabalho de software segue a primeira lei do ciclo: Não importa para onde se esteja indo, é sempre ladeira acima e contra o vento.”

Autor desconhecido

1. Projetos reais raramente seguem o fluxo sequencial que o modelo propõe. Embora o modelo linear possa conter iterações, ele o faz indiretamente. Como consequência, mudanças podem provocar confusão à medida que a equipe de projeto prossegue.
2. Frequentemente, é difícil para o cliente estabelecer explicitamente todas as necessidades. O modelo cascata requer isso e tem dificuldade para adequar a incerteza natural que existe no início de muitos projetos.
3. O cliente deve ter paciência. Uma versão operacional do(s) programa(s) não estará disponível antes de estarmos próximos do final do projeto. Um erro grave, se não detectado até o programa operacional ser revisto, pode ser desastroso.

Em uma interessante análise de projetos reais, Bradac [Bra94] descobriu que a natureza linear do ciclo de vida clássico conduz a “estados de bloqueio”, nos quais alguns membros da equipe do projeto têm de aguardar outros completarem tarefas dependentes. De fato, o tempo gasto na espera pode exceder o tempo gasto em trabalho produtivo! Os estados de bloqueio tendem a prevalecer no início e no final de um processo sequencial linear.

Hoje em dia, o trabalho de software tem um ritmo acelerado e está sujeito a uma cadeia de mudanças intermináveis (em características, funções e conteúdo de informações). O modelo cascata é frequentemente inapropriado para tal trabalho. Entretanto, ele pode servir como um modelo de processo útil em situações nas quais os requisitos são fixos e o trabalho deve ser realizado até sua finalização de forma linear.

PONTO-CHAVE

O modelo incremental libera uma série de versões, denominadas incrementais, que oferecem, progressivamente, maior funcionalidade para o cliente à medida que cada incremento é entregue.

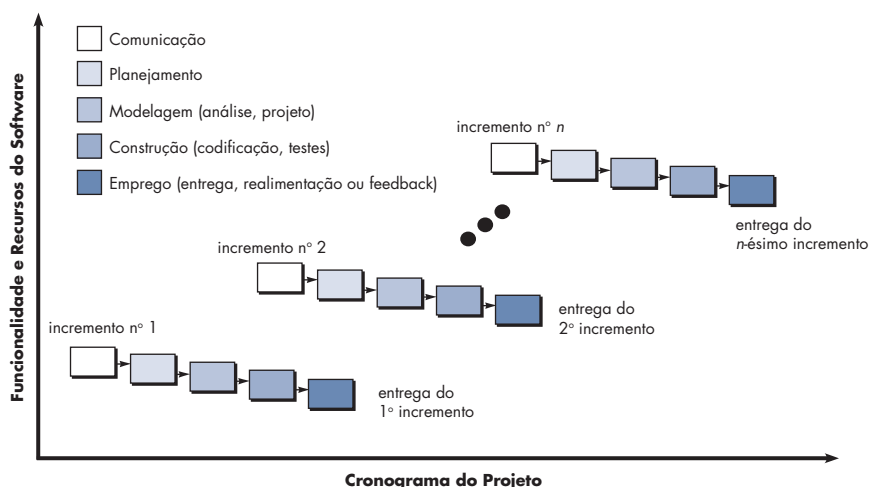
2.3.2 Modelos de processo incremental

Em várias situações, os requisitos iniciais do software são razoavelmente bem definidos, entretanto, devido ao escopo geral do trabalho de desenvolvimento, o uso de um processo puramente linear não é utilizado. Pode ser necessário o rápido fornecimento de um determinado conjunto funcional aos usuários, para somente após esse fornecimento, refinar e expandir sua funcionalidade em versões de software posteriores. Em tais casos, pode-se optar por um modelo de processo projetado para desenvolver o software de forma incremental.

O modelo *incremental* combina elementos dos fluxos de processos lineares e paralelos, discutidos na Seção 2.1. Na Figura 2.5, o modelo incremental aplica sequências lineares, de forma escalonada, à medida que o tempo vai avançando. Cada sequência linear gera “incrementais” (entregáveis/aprovados/liberados) do software [McD93] de maneira similar aos incrementais gerados por um fluxo de processos evolucionários (Seção 2.3.3).

FIGURA 2.5

O modelo incremental





Se o cliente precisa da entrega até uma determinada data impossível de ser atendida, sugira a entrega de um ou mais incrementos até a data e o restante do software (incrementos adicionais) posteriormente.

Por exemplo, um software de processamento de texto desenvolvido com o emprego do paradigma incremental, poderia liberar funções básicas de gerenciamento de arquivos, edição e produção de documentos no primeiro incremento; recursos mais sofisticados de edição e produção de documentos no segundo; revisão ortográfica e gramatical no terceiro; e, finalmente, recursos avançados de formatação (layout) de página no quarto incremento. Deve-se notar que o fluxo de processo para qualquer incremento pode incorporar o paradigma da prototipação.

Quando se utiliza um modelo incremental, frequentemente, o primeiro incremento é um *produto essencial*. Isto é, as os requisitos básicos são atendidos, porém, muitos recursos complementares (alguns conhecidos, outros não) ainda não são entregues. Esse produto essencial é utilizado pelo cliente (ou passa por uma avaliação detalhada). Como resultado do uso e/ou avaliação, é desenvolvido um planejamento para o incremento seguinte. O planejamento já considera a modificação do produto essencial para melhor se adequar às necessidades do cliente e à entrega de recursos e funcionalidades adicionais. Esse processo é repetido após a liberação de cada incremento, até que seja produzido o produto completo.

O modelo de processo incremental tem seu foco voltado para a entrega de um produto operacional com cada incremento. Os primeiros incrementos são versões seccionadas do produto final, mas eles realmente possuem capacidade para atender ao usuário e também oferecem uma plataforma para avaliação do usuário.⁸

O desenvolvimento incremental é particularmente útil nos casos em que não há pessoal disponível para uma completa implementação na época de vencimento do prazo estabelecido para o projeto. Os primeiros incrementos podem ser implementados com número mais reduzido de pessoal. Se o produto essencial for bem acolhido, então um pessoal adicional (se necessário) poderá ser acrescentado para implementar o incremento seguinte. Além disso, os incrementos podem ser planejados para administrar riscos técnicos. Por exemplo, um sistema importante pode exigir a disponibilidade de novo hardware que ainda está em desenvolvimento e cuja data de entrega é incerta. Poderia ser possível planejar incrementos iniciais de modo a evitar o uso desse hardware, possibilitando, portanto, a liberação de funcionalidade parcial aos usuários finais, sem um atraso excessivo.

PONTO-CHAVE

Os modelos de processo evolucionário produzem uma versão cada vez mais completa do software a cada iteração.

“Planeje para jogar algo fora. Você irá fazê-lo de qualquer maneira. Sua escolha consistirá em decidir se deve tentar ou não vender o que foi descartado aos clientes.”

Frederick P. Brooks

2.3.3 Modelos de processo evolucionário

Software, assim como todos sistemas complexos, evolui ao longo do tempo. Conforme o desenvolvimento do projeto avança, as necessidades de negócio e de produto mudam frequentemente, tornando inadequado seguir um planejamento em linha reta de um produto final. Prazos apertados, determinados pelo mercado, tornam impossível completar um produto de software abrangente, porém uma versão limitada tem de ser introduzida para aliviar e/ou atender às pressões comerciais ou da concorrência. Um conjunto do produto essencial ou das necessidades do sistema está bem compreendido, entretanto, detalhes de extensões do produto ou do sistema ainda devem ser definidos. Em situações como essa ou similares, faz-se necessário um modelo de processo que tenha sido projetado especificamente para desenvolver um produto que evolua ao longo do tempo.

Modelos evolucionários são iterativos. Apresentam características que possibilitam desenvolver versões cada vez mais completas do software. Nos parágrafos seguintes, são apresentados dois modelos comuns em processos evolucionários.

Prototipação. Frequentemente, o cliente define uma série de objetivos gerais para o software, mas não identifica, detalhadamente, os requisitos para funções e recursos. Em outros casos, o desenvolvedor encontra-se inseguro quanto à eficiência de um algoritmo, quanto à adaptabilidade de um sistema operacional ou quanto à forma em que deva ocorrer a interação homem/máquina. Em situações como essas, e em muitas outras, o *paradigma de prototipação* pode ser a melhor escolha de abordagem.

⁸ É importante notar que uma filosofia incremental também é utilizada em todos os modelos de processos “ágeis”, discutidos no Capítulo 3.



Quando seu cliente tiver uma necessidade legítima, mas sem a mínima ideia em relação a detalhes, faça um protótipo para uma primeira etapa.

Embora a prototipação possa ser utilizada como um modelo de processo isolado (stand-alone process) é mais comumente utilizada como uma técnica passível de ser implementada no contexto de qualquer um dos modelos de processo citados neste capítulo. Independentemente da forma como é aplicado, quando os requisitos estão obscuros, o paradigma da prototipação auxilia os interessados a compreender melhor o que está para ser construído.

O paradigma da prototipação (Figura 2.6) começa com a comunicação. Faz-se uma reunião com os envolvidos para definir os objetivos gerais do software, identificar quais requisitos já são conhecidos e esquematizar quais áreas necessitam, obrigatoriamente, de uma definição mais ampla. Uma iteração de prototipação é planejada rapidamente e ocorre a modelagem (na forma de um “projeto rápido”). Um projeto rápido se concentra em uma representação daqueles aspectos do software que serão visíveis aos usuários finais (por exemplo, o layout da interface com o usuário ou os formatos de exibição na tela).

O projeto rápido leva à construção de um protótipo, que é empregado e avaliado pelos envolvidos, que fornecerão um retorno (feedback), que servirá para aprimorar os requisitos. A iteração ocorre conforme se ajusta o protótipo às necessidades de vários interessados e, ao mesmo tempo, possibilita a melhor compreensão das necessidades que devem ser atendidas.

Na sua forma ideal, o protótipo atua como um mecanismo para identificar os requisitos do software. Caso seja necessário desenvolver um protótipo operacional, pode-se utilizar partes de programas existentes ou aplicar ferramentas (por exemplo, geradores de relatórios e gerenciadores de janelas) que possibilitem gerar rapidamente tais programas operacionais.

O que fazer com o protótipo quando este já serviu ao propósito descrito anteriormente? Brooks [Bro95] fornece uma resposta:

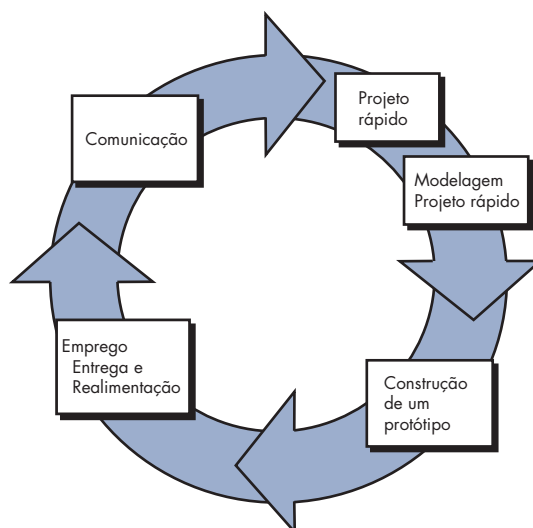
Na maioria dos projetos, o primeiro sistema dificilmente é utilizável. Pode estar muito lento, muito grande, estranho em sua utilização ou as três coisas juntas. Não há alternativa, a não ser começar de novo, ressentindo-se, porém, mais esperto (aprimorado), e desenvolver uma versão redesenhada na qual esses problemas são resolvidos.

O protótipo pode servir como “o primeiro sistema”. Aquele que Brooks recomenda que se jogue fora. Porém, essa pode ser uma visão idealizada. Embora alguns protótipos sejam construídos como “descartáveis”, outros são evolucionários, no sentido de que evoluem lentamente até se transformar no sistema real.

Tanto os interessados como os engenheiros de software gostam do paradigma da prototipação. Os usuários podem ter uma ideia prévia do sistema final, ao passo que os desenvolvedores

FIGURA 2.6

O paradigma da prototipação



passam a desenvolver algo imediatamente. Entretanto, a prototipação pode ser problemática pelas seguintes razões:



Resista à pressão de estender um protótipo grosseiro a um produto final. Quase sempre, como resultado, a qualidade fica comprometida.

1. Os interessados enxergam o que parece ser uma versão operacional do software, ignorando que o protótipo é mantido de forma não organizada e que, na pressa de fazer com que ele se torne operacional, não se considera a qualidade global do software, nem sua manutenção a longo prazo. Quando informados que o produto deve ser reconstruído para que altos níveis de qualidade possam ser mantidos, os interessados protestam e solicitam que “umas poucas correções” sejam feitas para tornar o protótipo um produto operacional. Frequentemente, a gerência do desenvolvimento de software aceita.
2. O engenheiro de software, com frequência, assume compromissos de implementação para conseguir que o protótipo entre em operação rapidamente. Um sistema operacional ou linguagem de programação inapropriados podem ser utilizados simplesmente porque se encontram à disposição e são conhecidos; um algoritmo ineficiente pode ser implementado simplesmente para demonstrar capacidade. Após um tempo, pode-se acomodar com tais escolhas e esquecer todas as razões pelas quais eram inapropriadas. Uma escolha longe da ideal acaba se tornando parte integrante do sistema.

Embora possam ocorrer problemas, a prototipação pode ser um paradigma efetivo para a engenharia de software. O segredo é definir as regras do jogo logo no início; isso significa que todos os envolvidos devem concordar que o protótipo é construído para servir como um mecanismo para definição de requisitos. Portanto, será descartado (pelo menos em parte) e o software final é arquitetado visando qualidade.

CASA SEGURA



Seleção de um Modelo de Processo, Parte 1

Cena: Sala de reuniões da equipe de engenharia de software na CPI Corporation, uma empresa (fictícia) que fabrica produtos de consumo para uso doméstico e comercial.

Participantes: Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software; Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; e Ed Robbins, membro da equipe de software.

A conversa:

Lee: Então, vamos recapitular. Passei algum tempo discutindo sobre a linha de produtos *CasaSegura*, da forma como a visualizamos no momento. Sem dúvida, temos um monte de trabalho a ser feito para simplesmente definir as coisas, mas eu gostaria que vocês comessem a pensar sobre como irão abordar a parte do software deste projeto.

Doug: Parece que temos sido bastante desorganizados em nossa abordagem sobre software no passado.

Ed: Eu não sei, Doug, nós sempre conseguimos entregar o produto.

Doug: É verdade, mas não sem grande sofrimento, e esse projeto parece ser maior e mais complexo do que qualquer outro que fizemos no passado.

Jamie: Não parece assim tão difícil, mas eu concordo... nossa abordagem “ad hoc” adotada para projetos anteriores não dará certo neste caso, principalmente se tivermos um cronograma muito apertado.

Doug (sorrindo): Quero ser um pouco mais profissional em nossa abordagem. Participei de um curso rápido na semana passada e aprendi bastante sobre engenharia de software... Bom conteúdo. Precisamos de um processo aqui.

Jamie (franzindo a testa): Minha função é desenvolver programas, não ficar mexendo em papéis.

Doug: Dê uma chance antes de me dizer não. Eis o que quero dizer. [Doug prossegue descrevendo a metodologia de processo descrita neste capítulo e os modelos de processo prescritivos apresentados até agora.]

Doug: Mas de qualquer forma, parece-me que um modelo linear não é adequado para nós... Ele assume que temos todos os requisitos antecipadamente e, conhecendo este lugar, isso é pouco provável.

Vinod: Isso mesmo, e parece orientado demais à tecnologia da informação... Provavelmente bom para construir um sistema de controle de estoque ou algo parecido, mas certamente não é adequado para o *CasaSegura*.

Doug: Eu concordo.

Ed: Essa abordagem de prototipação me parece OK. Bastante parecido com o que fazemos aqui.

Vinod: Isso é um problema. Estou preocupado que ela não nos dê estrutura suficiente.

Doug: Não é para se preocupar. Temos um monte de outras opções e quero que vocês escolham o que for melhor para a equipe e para o projeto.

Modelo Espiral. Originalmente proposto por Barry Boehm [Boe88], o *modelo espiral* é um modelo de processo de software evolucionário que acopla a natureza iterativa da prototipação com os aspectos sistemáticos e controlados do modelo cascata. Fornece potencial para o rápido desenvolvimento de versões cada vez mais completas do software. Boehm [Boe01a] descreve o modelo da seguinte maneira:

O modelo espiral de desenvolvimento é um gerador de *modelos de processos* dirigidos a riscos e é utilizado para guiar a engenharia de sistemas intensivos de software, que ocorre de forma concorrente e tem múltiplos envolvidos. Possui duas características principais que o distinguem. A primeira consiste em uma abordagem *cíclica voltada* para ampliar, incrementalmente, o grau de definição e a implementação de um sistema, enquanto diminui o grau de risco do mesmo. A segunda característica consiste em uma série de pontos âncora de controle para assegurar o comprometimento de interessados quanto à busca de soluções de sistema que sejam mutuamente satisfatórias e praticáveis.

Usando-se o modelo espiral, o software será desenvolvido em uma série de versões evolucionárias. Nas primeiras iterações, a versão pode consistir em um modelo ou em um protótipo. Já nas iterações posteriores, são produzidas versões cada vez mais completas do sistema que passa pelo processo de engenharia.

Um modelo espiral é dividido em um conjunto de atividades metodológicas definidas pela equipe de engenharia de software. Para fins ilustrativos, utilizam-se as atividades metodológicas genéricas discutidas anteriormente.⁹ Cada uma dessas atividades representa um segmento do caminho espiral ilustrado na Figura 2.7. Assim que esse processo evolucionário começa, a equipe de software realiza atividades indicadas por um circuito em torno da espiral no sentido horário, começando pelo seu centro. Os riscos (Capítulo 28) são considerados à medida que cada revolução é realizada. Pontos âncora de controle — uma combinação de produtos de trabalho e condições que são satisfeitas ao longo do trajeto da espiral — são indicados para cada passagem evolucionária.

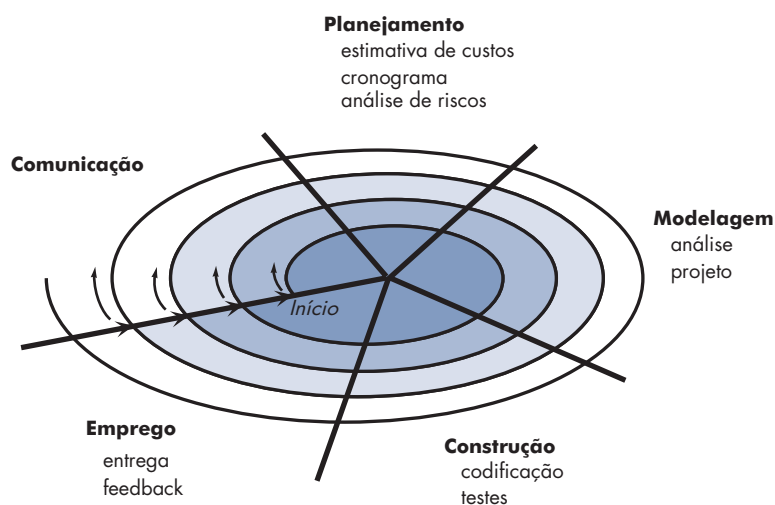
O primeiro circuito em volta da espiral pode resultar no desenvolvimento de uma especificação de produto; passagens subsequentes em torno da espiral podem ser usadas para desenvolver um protótipo e, então, progressivamente, versões cada vez mais sofisticadas do software. Cada passagem pela região de planejamento resulta em ajustes no planejamento do projeto.

PONTO-CHAVE

O modelo espiral pode ser adaptado para ser aplicado ao longo de todo o ciclo de vida de uma aplicação, desde o desenvolvimento de conceitos até sua manutenção.

FIGURA 2.7

Modelo espiral típico



⁹ O modelo espiral discutido nesta seção é uma variação do modelo proposto por Boehm. Para mais informações sobre o modelo espiral original, consulte [Boe88]. Material mais recente sobre o modelo espiral de Boehm pode ser encontrado em [Boe98].

WebRef

Informações úteis sobre o modelo espiral podem ser obtidas em: www.sei.cmu.edu/publications/documents/00.reports/00sr008.html.



Se a gerência quiser um desenvolvimento com orçamento fixo (geralmente uma péssima ideia), a espiral pode ser um problema. À medida que cada circuito for realizado, o custo do projeto será repetidamente revisado.

"Estou tão perto, mas apenas o amanhã guia o meu caminho."

Dave Matthews Band

Custo e cronograma são ajustados de acordo com o feedback (a realimentação) obtido do cliente após entrega. Além disso, o gerente de projeto faz um ajuste no número de iterações planejadas para completar o software.

Diferente de outros modelos de processo, que terminam quando o software é entregue, o modelo espiral pode ser adaptado para ser aplicado ao longo da vida do software. Portanto, o primeiro circuito em torno da espiral pode representar um "projeto de desenvolvimento de conceitos" que começa no núcleo da espiral e continua por várias iterações¹⁰, até que o desenvolvimento de conceitos esteja completo. Se o conceito for desenvolvido para ser um produto final, o processo prossegue pela espiral pelas "bordas" e um "novo projeto de desenvolvimento de produto" se inicia. O novo produto evoluirá, passando por uma série de iterações, em torno da espiral. Posteriormente, uma volta em torno da espiral pode ser usada para representar um "projeto de aperfeiçoamento do produto". Em sua essência, a espiral, caracterizada dessa maneira, permanece em operação até que o software seja retirado. Há casos em que o processo fica inativo, porém, toda vez que uma mudança é iniciada, começa no ponto de partida apropriado (por exemplo, aperfeiçoamento do produto).

O modelo espiral é uma abordagem realista para o desenvolvimento de sistemas e de software em larga escala. Pelo fato de o software evoluir à medida que o processo avança, o desenvolvedor e o cliente compreendem e reagem melhor aos riscos em cada nível evolucionário. Esse modelo usa a prototipação como mecanismo de redução de riscos e, mais importante, torna possível a aplicação da prototipação em qualquer estágio do processo evolutivo do produto. Mantém a abordagem em etapas, de forma sistemática, sugerida pelo ciclo de vida clássico, mas a incorpora em uma metodologia iterativa que reflete mais realisticamente o mundo real. O modelo espiral requer consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado apropriadamente, reduz os riscos antes de se tornarem problemáticos.

Mas como outros paradigmas, esse modelo não é uma panaceia. Pode ser difícil convencer os clientes (particularmente em situações contratuais) de que a abordagem evolucionária é controlável. Ela exige considerável especialização na avaliação de riscos e depende dessa especialização para seu sucesso. Se um risco muito importante não for descoberto e administrado, indubitavelmente ocorrerão problemas.

CASA SEGURA



Seleção de um modelo de processo, Parte 2

Cena: Sala de reuniões do grupo de engenharia de software na CPI Corporation, empresa (fictícia) que fabrica produtos de consumo de uso doméstico e comercial.

Participantes: Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software; Vinod e Jamie, membros da equipe de engenharia de software.

Conversa: [Doug descreve as opções do processo evolucionário.]

Jamie: Agora estou vendo algo de que gosto. Faz sentido uma abordagem incremental e eu realmente gosto do fluxo dessa coisa de modelo espiral. Isso tem a ver com a realidade.

Vinod: Concordo. Entregamos um incremento, aprendemos com o feedback do cliente, replanejamos e, então, entregamos

outro incremento. Também se encaixa na natureza do produto. Podemos colocar alguma coisa no mercado rapidamente, ter algo no mercado e, depois, acrescentar funcionalidade a cada versão, digo, incremento.

Lee: Espere um pouco, você disse que reformulamos o plano a cada volta na espiral, Doug? Isso não é tão legal; precisamos de um plano, um cronograma e temos de nos ater a ele.

Doug: Essa linha de pensamento é antiga, Lee. Como o pessoal disse, temos de mantê-lo real. Acho que é melhor ir ajustando o planejamento na medida em que formos aprendendo mais e as mudanças sejam solicitadas. É muito mais realista. Para que serve um plano se não refletir a realidade?

Lee (franzindo a testa): Suponho que esteja certo, porém... A alta gerência não vai gostar disso... Querem um plano fixo.

Doug (sorrindo): Então, você terá que reeducá-los, meu amigo.

¹⁰ As setas que apontam para dentro, ao longo do eixo, separando a região de **emprego** da região de **comunicação**, indicam potencial para iteração local ao longo do mesmo trajeto da espiral.

2.3.4 Modelos concorrentes

O *modelo de desenvolvimento concorrente*, algumas vezes denominado *engenharia concorrente*, possibilita à equipe de software representar elementos concorrentes e iterativos de qualquer um dos modelos de processos descritos neste capítulo. Por exemplo, a atividade de modelagem definida para o modelo espiral é realizada invocando uma ou mais das seguintes ações de engenharia de software: prototipagem, análise e projeto.¹¹



O modelo concorrente, com frequência, é mais adequado para projetos de engenharia de produto nos quais diferentes equipes de engenharia estão envolvidas.

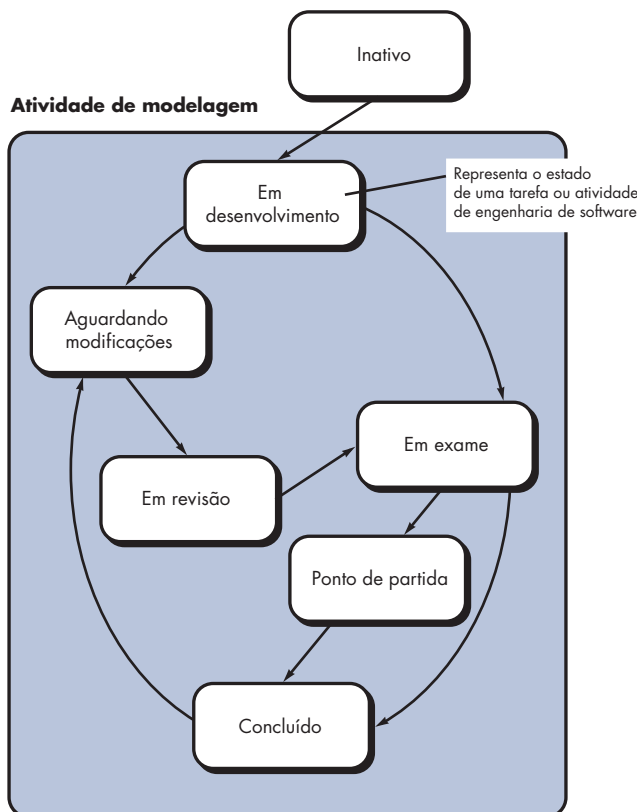
A Figura 2.8 mostra um esquema de uma atividade da engenharia de software, dentro da atividade de modelagem, usando uma abordagem de modelagem concorrente. A atividade — **modelagem** — poderia estar em qualquer um dos estados¹² observados em qualquer instante determinado. Similarmente, outras atividades, ações ou tarefas (por exemplo, **comunicação** ou **construção**) podem ser representadas de maneira análoga. Todas as atividades de engenharia de software existem concorrentemente, porém estão em diferentes estados.

Por exemplo, no início de um projeto, a atividade de comunicação (não mostrada na figura) completou sua primeira iteração e se encontra no estado **aguardando modificações**. A atividade de modelagem (que se encontrava no estado **inativo** enquanto a comunicação inicial era completada, agora faz uma transição para o estado **em desenvolvimento**. Se, entretanto, o cliente indicar que mudanças nos requisitos devem ser feitas, a atividade de modelagem passa do estado **em desenvolvimento** para o estado **aguardando modificações**.

A modelagem concorrente define uma série de eventos que irão disparar transições de estado para estado para cada uma das atividades, ações ou tarefas da engenharia de software.

FIGURA 2.8

Um elemento do modelo de processo concorrente



¹¹ Deve-se notar que a análise e o projeto são tarefas complexas que requerem discussão substancial. A Parte 2 deste livro considera esses tópicos em detalhe.

¹² Um *estado* é algum modo externamente observável do comportamento.

“Em todo processo há um cliente, pois, sem cliente, um processo deixa de ter sentido.”

**V. Daniel
Hunt**

Por exemplo, durante estágios de projeto iniciais (uma ação de engenharia de software importante que ocorre durante a atividade de modelagem), uma inconsistência no modelo de requisitos não é descoberta. Isso gera o evento *correção do modelo de análise*, que irá disparar a ação de análise de requisitos, passando do estado **concluído** para o estado **aguardando modificações**.

A modelagem concorrente se aplica a todos os tipos de desenvolvimento de software e fornece uma imagem precisa do estado atual de um projeto. Em vez de limitar as atividades, ações e tarefas da engenharia de software a uma sequência de eventos, ela define uma rede de processos. Cada atividade, ação ou tarefa na rede existe simultaneamente com outras atividades, ações ou tarefas. Eventos gerados em um ponto da rede de processos disparam transições entre os estados.

2.3.5 Um comentário final sobre processos evolucionários

Como já foi mencionado anteriormente, os softwares modernos são caracterizados por contínuas modificações, prazos muito apertados e por uma ênfase na satisfação do cliente-usuário. Em muitos casos, o tempo de colocação de um produto no mercado é o requisito mais importante a ser gerenciado. Se o momento oportuno de entrada no mercado for perdido, o projeto de software pode ficar sem sentido.¹³

Os modelos de processo evolucionário foram concebidos para tratar dessas questões e, mesmo assim, como uma classe genérica de modelos de processo, apresentam seus pontos fracos. Esses pontos fracos foram resumidos por Nogueira e seus colegas [Nog00]:

Apesar dos inquestionáveis benefícios proporcionados pelos processos de software evolucionários, temos algumas preocupações. A primeira delas é que a prototipação [e outros processos evolucionários mais sofisticados] traz um problema para o planejamento do projeto devido ao número incerto de ciclos necessários para construir o produto. A maior parte das técnicas de gerenciamento e de estimativas do projeto baseia-se em layouts lineares das atividades, o que faz com que não se adequem completamente.

Em segundo lugar, os processos de software evolucionários não estabelecem a velocidade máxima da evolução. Se as evoluções ocorrerem numa velocidade excessivamente rápida, sem um período de acomodação, é certo que o processo cairá no caos. Por outro lado, se a velocidade for muito lenta, então a produtividade poderia ser afetada...

Em terceiro lugar, os processos de software devem manter seu foco mais na flexibilidade e na extensibilidade do que na alta qualidade. Essa afirmação soa assustadora. Entretanto, deve-se priorizar mais a velocidade de desenvolvimento do que a do desenvolvimento com zero defeito. Prolongar o desenvolvimento em busca da alta qualidade pode resultar em entrega tardia do produto, quando o nicho de oportunidade já desapareceu. Essa mudança de paradigma é imposta pela concorrência em situações em que se está à beira do caos.

Realmente, um processo de software priorizando flexibilidade, extensibilidade e velocidade de desenvolvimento, acima da alta qualidade, soa assustador. Ainda assim, essa ideia foi proposta por uma série de renomados especialistas em engenharia de software (como, por exemplo, [You95], [Bac97]).

O objetivo dos modelos evolucionários é desenvolver software de alta qualidade¹⁴ de modo iterativo ou incremental. Entretanto, é possível usar um processo evolucionário para enfatizar a flexibilidade, a extensibilidade e a velocidade do desenvolvimento. O desafio para as equipes de software e seus gerentes será estabelecer um equilíbrio apropriado entre esses parâmetros críticos de projeto e produto e a satisfação dos clientes (o árbitro final da qualidade de um software).

¹³ É importante notar, entretanto, que ser o primeiro a chegar ao mercado não é sinônimo de sucesso. De fato, muitos produtos de software bem-sucedidos foram o segundo ou até mesmo o terceiro a chegar ao mercado (aprendendo com os erros dos outros que o antecederam).

¹⁴ Nesse contexto, a qualidade de software é definida de forma bastante abrangente para englobar não apenas a satisfação dos clientes, como também uma série de critérios técnicos, discutidos nos Capítulos 14 e 16.

2.4 MODELOS DE PROCESSO ESPECIALIZADO

Os modelos de processo especializado levam em conta muitas das características de um ou mais dos modelos tradicionais apresentados nas seções anteriores. Tais modelos tendem a ser aplicados quando se opta por uma abordagem de engenharia de software especializada ou definida de forma restrita.¹⁵

2.4.1 Desenvolvimento baseado em componentes

WebRef

Informações úteis sobre desenvolvimento baseado em componentes podem ser encontradas em: www.cbd-hq.com.

Componentes de software comercial de prateleira ou COTS (sigla para Commercial Off-The-Shelf), desenvolvidos por vendedores que os oferecem como produtos, disponibilizam a funcionalidade almejada juntamente com as bem definidas interfaces, sendo que essas interfaces permitem que o componente seja integrado ao software a ser desenvolvido. O *modelo de desenvolvimento baseado em componentes* incorpora muitas das características do modelo espiral. É evolucionário em sua natureza [Nie92], demandando uma abordagem iterativa para a criação de software. O modelo de desenvolvimento baseado em componentes desenvolve aplicações a partir de componentes de software pré-empacotados.

As atividades de modelagem e construção começam com a identificação de possíveis candidatos a componentes. Esses componentes podem ser projetados como módulos de software convencionais, como classes orientadas a objeto ou pacotes¹⁶ de classes. Independentemente da tecnologia usada para criar os componentes, o modelo de desenvolvimento baseado em componentes incorpora as seguintes etapas (implementadas usando-se uma abordagem evolucionária):

1. Produtos baseados em componentes disponíveis são pesquisados e avaliados para o campo de aplicação em questão.
2. Itens de integração de componentes são considerados.
3. Uma arquitetura de software é projetada para acomodar os componentes.
4. Os componentes são integrados na arquitetura.
5. Testes completos são realizados para assegurar funcionalidade adequada.

O modelo de desenvolvimento baseado em componentes conduz ao reuso do software e a reusabilidade proporciona uma série de benefícios mensuráveis aos engenheiros de software. A equipe de engenharia de software pode conseguir uma redução no tempo do ciclo de desenvolvimento, bem como uma redução no custo do projeto, caso a reutilização de componentes se torne parte de sua cultura. O desenvolvimento baseado em componentes é discutido de forma mais detalhada no Capítulo 10.

2.4.2 O modelo de métodos formais

O *modelo de métodos formais engloba* um conjunto de atividades que conduzem à especificação matemática formal do software. Os métodos formais possibilitam especificar, desenvolver e verificar um sistema baseado em computador através da aplicação de uma notação matemática rigorosa. Uma variação dessa abordagem, chamada *engenharia de software “cleanroom” (sala limpa/leve/nítida)* [Mil87, Dye92], é aplicada atualmente por algumas organizações de desenvolvimento de software.

Quando são utilizados métodos formais (Capítulo 21) durante o desenvolvimento, estes oferecem um mecanismo que elimina muitos dos problemas difíceis de ser superados com o uso

¹⁵ Em alguns casos, esses modelos de processo especializado podem ser mais bem definidos como um conjunto de técnicas, ou uma “metodologia”, para alcançar uma meta de desenvolvimento de software específica. Entretanto, eles realmente implicam um processo.

¹⁶ Conceitos de orientação a objetos são discutidos no Apêndice 2 e são usados ao longo da Parte 2 deste livro. Nesse contexto, uma classe engloba um conjunto de dados e os procedimentos que processam os mesmos. Pacote de classes é um conjunto de classes relativas que operam juntas para alcançar algum resultado final.

de outros paradigmas de engenharia de software. Ambiguidade, incompletude e inconsistência podem ser descobertas e corrigidas mais facilmente — não por meio de uma revisão local, mas devido à aplicação de análise matemática. Quando são utilizados métodos formais durante o projeto, servem como base para verificar a programação e, portanto, possibilitam que se descubra e se corrija erros que, de outra forma, poderiam passar despercebidos.

Embora não seja uma abordagem predominante, o modelo de métodos formais oferece a promessa de software sem defeitos. No entanto, foram mencionados motivos para preocupação a respeito de sua aplicabilidade em um ambiente de negócios:

- Atualmente, o desenvolvimento de modelos formais consome muito tempo e dinheiro.
- Pelo fato de poucos desenvolvedores de software possuírem formação e experiência necessárias (background) para aplicação dos métodos formais, é necessário treinamento extensivo.
- É difícil usar os modelos como um meio de comunicação com clientes tecnicamente despreparados (não sofisticados tecnicamente).

Apesar de tais preocupações, a abordagem de métodos formais tem conquistado adeptos entre os desenvolvedores de software que precisam desenvolver software com fator crítico de segurança (como, por exemplo, os desenvolvedores de sistemas aviônicos para aeronaves e equipamentos médicos), bem como entre desenvolvedores que sofreriam pesadas sanções econômicas se ocorressem erros no software.

2.4.3 Desenvolvimento de software orientado a aspectos

Independentemente do processo de software escolhido, os desenvolvedores de software complexos, invariavelmente, implementam um conjunto de recursos, funções e conteúdo localizados. Essas características de software localizadas são modeladas como componentes (por exemplo, classes orientadas a objetos) e, em seguida, construídas dentro do contexto da arquitetura do sistema. À medida que os modernos sistemas baseados em computadores se tornam mais sofisticados (e complexos), certas *restrições* — propriedades exigidas pelo cliente ou áreas de interesse técnico — se estendem por toda a arquitetura. Algumas restrições são propriedades de alto nível de um sistema (por exemplo, segurança, tolerância a falhas). Outras afetam funções (por exemplo, a aplicação de regras de negócio), sendo que outras são sistêmicas (por exemplo, sincronização de tarefas ou gerenciamento de memória).

Quando restrições cruzam múltiplas funções, recursos e informações do sistema, elas são, frequentemente, denominadas *restrições cruzadas*. Os *requisitos de aspectos* definem as restrições cruzadas que têm um impacto por toda a arquitetura de software. O desenvolvimento de software orientado a aspectos AOSD, *Aspect-Oriented Software Development*), com frequência conhecido como programação orientada a aspectos (AOP, *Aspect-Oriented Programming*), é um paradigma de engenharia de software relativamente novo que oferece uma abordagem metodológica e de processos para definir, especificar, projetar e construir *aspectos* — “mecanismos além das sub-rotinas e herança para localizar a expressão de uma restrição cruzada” [Elr01].

Grundy [Gru02] discute ainda mais os aspectos no contexto do que ele denomina engenharia de componentes orientada a aspectos (AOCE, *aspect-oriented component engineering*):

A AOCE usa um conceito de fatias horizontais através de componentes de software decompostos verticalmente, chamados “aspectos”, para caracterizar propriedades funcionais ou não funcionais cruzadas dos componentes. Aspectos sistêmicos, comuns, incluem interfaces com o usuário, trabalho colaborativo, distribuição, persistência, gerenciamento de memória, processamento de transações, segurança, integridade e assim por diante. Os componentes podem fornecer ou requerer um ou mais “detalhes de aspecto” relativo a um determinado aspecto, tais como um mecanismo de visualização, exequibilidade extensível e gênero de interface (aspectos da interface com o usuário); geração de eventos, transporte e recebimento (aspectos de distribuição); armazenamento/recuperação e indexação de dados (aspectos de persistência); autenticação, codificação e direitos de acesso (aspectos de segurança); atomicidade

? Se métodos formais são capazes de demonstrar correção de software, por que não são amplamente utilizados?

WebRef

Uma ampla gama de recursos e informações sobre AOP pode ser encontrada em: aosd.net.

PONTO-CHAVE

A AOSD define “aspectos” representando restrições do cliente que cruzam várias funções, recursos e informações do sistema.

de transações, controle de concorrência e estratégia de entrada no sistema (aspectos transacionais) e assim por diante. Cada detalhe de aspecto possui uma série de propriedades relativas a características funcionais e não funcionais do detalhe do aspecto.

Um processo distinto orientado a aspectos ainda não atingiu sua maturação. Entretanto, é provável que um processo desses irá adotar características tanto dos modelos de processo evolucionário quanto de concorrente. O modelo evolucionário é apropriado quando os aspectos são identificados e então construídos. A natureza paralela do desenvolvimento concorrente é essencial, porque os aspectos são criados independentemente de componentes de software localizado e, apesar disso, os aspectos têm um impacto direto sobre esses componentes. Portanto, é essencial instanciar comunicação assíncrona entre as atividades de processos de software aplicadas na engenharia e construção de aspectos e componentes.

Uma discussão detalhada sobre desenvolvimento de software orientado a aspectos é mais bem colocada em livros dedicados ao assunto. Se tiver mais interesse, consulte [Saf08], [Cla05], [Jac04] e [Gra03].

FERRAMENTAS DO SOFTWARE



Gerenciamento de processos

Objetivo: ajudar na definição, execução e gerenciamento de modelos de processos prescritivos.

Mecânica: as ferramentas de gerenciamento de processo possibilitam que uma organização ou equipe de software definam um modelo de processo de software completo (atividades metodológicas, ações, tarefas, pontos de verificação para garantia da qualidade, pontos de controle (marcos) e artefatos de software). Além disso, tais ferramentas acabam fornecendo um mapeamento (guia), conforme os engenheiros de software realizam o trabalho técnico, e também propiciam um modelo para os gerentes, os quais têm o dever de acompanhar e controlar o processo de software.

Ferramentas Representativas:¹⁷ o GDPA, um conjunto de ferramentas para definição de processos de pesquisa, desen-

volvido na Universidade de Bremen, na Alemanha (www.informatik.uni-bremen.de/uniform/gdpa/home.htm), fornece uma grande quantidade de funções de gerenciamento e modelagem de processos.

○ *SpeedDev*, desenvolvido pela SpeedDev Corporation (www.speeddev.com) engloba um conjunto de ferramentas para definição de processo, gerenciamento de requisitos, resolução de itens, planejamento e acompanhamento de projetos.

○ *ProVision BPMx*, desenvolvido pela Proforma (www.proformacorp.com), é representante de muitas ferramentas que auxiliam na definição de processo e automação de fluxo de trabalho.

Uma lista valiosa de diversas ferramentas diferentes associadas ao processo de software, pode ser encontrada no endereço www.processwave.net/Links/tool_links.htm.

2.5 O PROCESSO UNIFICADO

No livro que deu origem ao *Processo Unificado*, Ivar Jacobson, Grady Booch e James Rumbaugh [Jac99] discutem a necessidade de um processo de software “dirigido a casos de uso, centrado na arquitetura, iterativo e incremental” ao afirmarem:

Hoje em dia, a tendência do software é no sentido de sistemas maiores e mais complexos. Isso se deve, em parte, ao fato de que os computadores tornam-se mais potentes a cada ano, levando os usuários a ter uma expectativa maior em relação a eles. Essa tendência também foi influenciada pelo uso crescente da Internet para troca de todos os tipos de informação... Nosso apetite por software cada vez mais sofisticado aumenta à medida que tomamos conhecimento de uma versão do produto para a seguinte, como o produto pode ser aperfeiçoado. Queremos software que seja mais e mais adaptado a nossas necessidades, mas isso, por sua vez, simplesmente torna o software mais complexo. Em suma, queremos cada vez mais.

De certa forma, o Processo Unificado é uma tentativa de aproveitar os melhores recursos e características dos modelos tradicionais de processo de software, mas caracterizando-os de modo a implementar muitos dos melhores princípios do desenvolvimento ágil de software (Capítulo 3). O Processo Unificado reconhece a importância da comunicação com o cliente e de métodos racionalizados (sequencializados) para descrever a visão do cliente sobre um sistema (os

casos de uso¹⁷). Ele enfatiza o importante papel da arquitetura de software e “ajuda o arquiteto a manter o foco nas metas corretas, tais como compreensibilidade, confiança em mudanças futuras e reutilização” [Jac99]. Ele sugere um fluxo de processo iterativo e incremental, proporcionando a sensação evolucionária que é essencial no desenvolvimento de software moderno.

2.5.1 Breve histórico

Durante o início dos anos 1990, James Rumbaugh [Rum91], Grady Booch [Boo94] e Ivar Jacobson [Jac92] começaram a trabalhar em um “método unificado” que combinaria as melhores características de cada um de seus métodos individuais de análise e projeto orientados a objetos e adotaram características adicionais propostas por outros especialistas (por exemplo, [Wir90]) em modelagem orientada a objetos. O resultado foi a UML — uma *linguagem de modelagem unificada* que contém uma notação robusta para a modelagem e o desenvolvimento de sistemas orientados a objetos. Por volta de 1997, a UML tornou-se um padrão de fato da indústria em termos de desenvolvimento de software orientado a objetos.

A UML é usada ao longo da Parte 2 deste livro para representar tanto modelos de projeto quanto de requisitos. O Apêndice 1 apresenta um tutorial introdutório para aqueles que não estão familiarizados com as regras básicas de notações e de modelagem da UML. Uma apresentação completa da UML fica reservada a livros-texto dedicados ao assunto. Livros recomendados estão relacionados no Apêndice 1.

A UML forneceu a tecnologia necessária para dar suporte à prática de engenharia de software orientada a objetos, mas não ofereceu a metodologia de processo para orientar as equipes de projeto na aplicação da tecnologia. Ao longo de poucos anos que se seguiram, Jacobson, Rumbaugh e Booch desenvolveram o *Processo Unificado*, uma metodologia para engenharia de software orientada a objetos usando a UML. Hoje em dia, o Processo Unificado (PU ou UP, Unified Process) e a UML são amplamente utilizados em projetos orientados a objetos de todos os tipos. O modelo incremental e iterativo proposto pelo PU pode e deve ser adaptado para atender necessidades de projeto específicas.

2.5.2 Fases do processo unificado¹⁸

No início deste capítulo, foram apresentadas cinco atividades metodológicas genéricas e argumentos afirmando que elas poderiam ser usadas para descrever qualquer modelo de processo de software. O Processo Unificado não é nenhuma exceção. A Figura 2.9 descreve as “fases” do PU e as relaciona com as atividades genéricas que foram discutidas no Capítulo 1 e anteriormente neste capítulo.

A *fase de concepção (Inception)* do PU envolve tanto a atividade de comunicação com o cliente como a de planejamento. Colaborando com os interessados, identificam-se as necessidades de negócio para o software; propõe-se uma arquitetura rudimentar para o sistema e se desenvolve um planejamento para a natureza iterativa e incremental do projeto decorrente. Requisitos de negócio fundamentais são descritos por meio de um conjunto de casos práticos preliminares (Capítulo 5), descrevendo quais recursos e funções cada categoria principal de usuário deseja. Até esse ponto, a arquitetura nada mais é do que um esquema provisório dos principais subsistemas e da função e dos recursos que os compõem. Posteriormente, a arquitetura será refinada e expandida para um conjunto de modelos que representarão visões diferentes do sistema. O planejamento identifica recursos, avalia os principais riscos, define um cronograma e estabelece uma base para as fases que serão aplicadas à medida que o incremento de software é desenvolvido.

A *fase de elaboração* envolve atividades de comunicação e modelagem do modelo de processo genérico (Figura 2.9). A elaboração refina e expande os casos práticos preliminares, desenvol-

PONTO-CHAVE

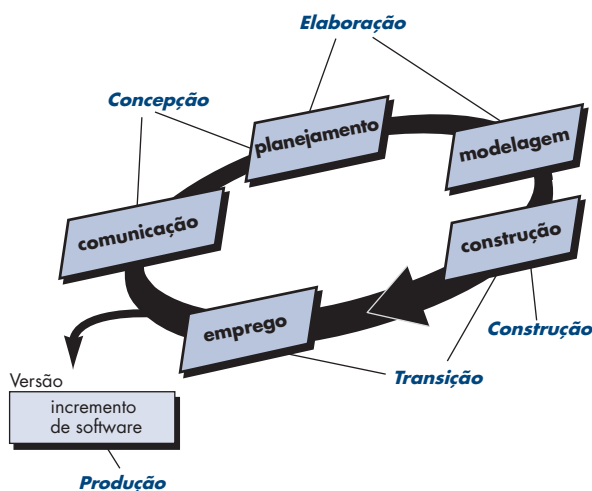
Em seu intento, as fases do Processo Unificado são similares às atividades metodológicas genéricas definidas neste livro.

¹⁷ Um *caso de uso* (Capítulo 5) é uma narrativa textual ou modelo que descreve uma função ou recurso de um sistema do ponto de vista do usuário. Um caso de uso é escrito pelo usuário e serve como base para criação de um modelo de requisitos mais amplo.

¹⁸ O Processo Unificado é, algumas vezes, chamado de *Processo Unificado Racional* RUP, Rational Unified Process) em homenagem a Rational Corporation (posteriormente adquirida pela IBM), um dos primeiros contribuidores para o desenvolvimento e refinamento do PU e um desenvolvedor de ambientes completos (ferramentas e tecnologia) que dão suporte ao processo.

FIGURA 2.9

O Processo Unificado



vidos como parte da fase de concepção, e amplia a representação da arquitetura, incluindo cinco visões diferentes do software: modelo de caso prático, modelo de requisitos, modelo de projeto, modelo de implementação e modelo de emprego. Em alguns casos, a elaboração gera uma “base de arquitetura executável” [Arl02], consistindo num sistema executável “de degustação”.¹⁹ Essa base demonstra a viabilidade da arquitetura, mas não oferece todos os recursos e funções necessárias para usar o sistema. Além disso, no auge da fase de elaboração, o plano é revisado cuidadosamente para assegurar que escopo, riscos e datas de entrega permaneçam razoáveis. Normalmente, as modificações no planejamento são feitas nesta oportunidade.

WebRef

Uma interessante abordagem sobre o PU, dentro do contexto de desenvolvimento ágil, poderá ser encontrada em www.ambysoft.com/unifiedprocess/agileUP.html.

A *fase de construção* do PU é idêntica à atividade de construção definida para o processo de software genérico. Tendo como entrada (input) o modelo de arquitetura, a fase de construção desenvolve ou adquire componentes de software; esses componentes farão com que cada caso prático (de uso) se torne operacional para os usuários finais. Para tanto, os modelos de requisitos e de projeto, iniciados durante a fase de elaboração, são completados para refletir a versão final do incremento de software. Então, implementa-se, no código-fonte, todos os recursos e funções necessárias e exigidas para o incremento de software (isto é, para a versão). À medida que os componentes estão sendo implementados, desenvolve-se e executam-se testes de unidades²⁰ para cada um deles. Além disso, realizam-se atividades de integração (montagem de componentes e testes de integração). Os casos práticos são usados para obter um pacote de testes de aceitação, executados antes do início da fase seguinte do PU.

A *fase de transição* do PU abarca os últimos estágios da atividade da construção genérica e a primeira parte da atividade de emprego genérico: entrega e realimentação (feedback). Entrega-se o software aos usuários finais para testes beta e o feedback dos usuários relata defeitos e mudanças necessárias. Além disso, a equipe de software elabora material com as informações de apoio (por exemplo, manuais para o usuário, guias para resolução de problemas, procedimentos de instalação) que são necessárias para lançamento da versão. Na conclusão da fase de transição, o incremento torna-se uma versão do software utilizável.

A *fase de produção* do PU coincide com a atividade de emprego do processo genérico. Durante essa fase, monitora-se o uso contínuo do software, disponibiliza-se suporte para o ambiente (infraestrutura) operacional, realiza-se e avalia-se relatórios de defeitos e solicitações de mudanças.

¹⁹ É importante observar que a base da arquitetura não é um protótipo, já que ela não é descartada. Ao contrário, a base ganha corpo durante a fase seguinte do PU.

²⁰ Uma discussão extensiva de testes de software (inclusive *testes de unidades*) é apresentada nos Capítulos 17 a 20.

É provável que, ao mesmo tempo em que as fases de construção, transição e produção estejam sendo conduzidas, já se tenha iniciado o incremento de software seguinte. Isso significa que as cinco fases do PU não ocorrem em sequência, mas sim de forma concorrente e escalonada.

Um fluxo de trabalho de engenharia de software é distribuído ao longo de todas as fases do PU. Nesse contexto, um *fluxo de trabalho* é análogo a um conjunto de tarefas (descrito anteriormente neste capítulo), isto é, identifica as tarefas para realizar uma importante ação de engenharia de software e os artefatos produzidos como consequência da finalização de tarefas com êxito. Deve-se notar que nem toda tarefa identificada para um fluxo de trabalho do PU é conduzida em todos os projetos de software. A equipe adapta o processo (ações, tarefas, subtarefas e artefatos de software) para ficar de acordo com suas necessidades.

2.6 MODELOS DE PROCESSO PESSOAL E DE EQUIPE

“Pessoas bem-sucedidas desenvolveram, simplesmente, o hábito de realizar coisas que as fracassadas não irão fazer.”

Dexter Yager

O melhor processo de software é aquele próximo às pessoas que realizarão o trabalho. Se um modelo de processo de software for desenvolvido em nível corporativo ou organizacional, ele apenas será efetivo se for aberto a significativas adaptações a fim de atender às necessidades da equipe de projeto (aquela que está efetivamente realizando o trabalho de engenharia de software). Num cenário ideal, você desenvolveria um processo que melhor se adequasse às suas necessidades e, simultaneamente, atendessem às necessidades mais amplas da equipe e da organização. De forma alternativa, a própria equipe pode criar seu próprio processo e, ao mesmo tempo, atender às necessidades mais específicas dos indivíduos e às necessidades mais amplas da organização. Watts Humphrey ([Hum97] e [Hum00]) afirmam que é possível criar um “processo de software pessoal” e/ou um “processo de software da equipe”. Ambos requerem trabalho árduo, treinamento e coordenação, mas são alcançáveis.²¹

2.6.1 Processo de Software Pessoal (PSP)

Todo desenvolvedor utiliza algum processo para construir software. Esse processo pode ser nebuloso ou específico; pode mudar diariamente; não ser eficiente, efetivo ou bem-sucedido; porém, um “processo” realmente existe. Watts Humphrey [Hum97] sugere que a fim de modificar um processo pessoal não efetivo, um indivíduo deve passar por quatro fases, cada uma exigindo treinamento e orquestração cuidadosa. O *Processo de Software Pessoal* (sigla PSP, *Personal Software Process*) enfatiza a medição pessoal, tanto do artefato de software gerado quanto da qualidade resultante dele. Além disso, responsabiliza o profissional pelo planejamento de projetos (por exemplo, estimativa de custos e cronograma) e lhe dá poder para controlar a qualidade de todos os artefatos de software desenvolvidos. O modelo PSP define cinco atividades estruturais:

Planejamento. Essa atividade isola os requisitos e desenvolve as estimativas de porte e de recursos. Além disso, faz-se uma estimativa dos defeitos (o número de defeitos estimado para o trabalho). Registram-se todas as métricas em formulários ou planilhas. Finalmente, identificam-se as tarefas de desenvolvimento e faz-se um cronograma para o projeto.

Projeto de alto nível. Desenvolvem-se especificações externas para cada componente a ser construído e elabora-se um projeto de componentes. Quando há incerteza, constroem-se protótipos. Todos os problemas são registrados e localizados.

Revisão de projeto de alto nível. Aplicam-se métodos de verificação formais (Capítulo 21) para revelar erros no projeto. Métricas são mantidas para todos os resultados de trabalho e tarefas importantes.

Desenvolvimento. O projeto em nível de componentes é refinado e revisado. Código é gerado, revisado, compilado e testado. Métricas são mantidas para todos os resultados de trabalho e tarefas importantes.

WebRef

Uma grande quantidade de recursos para PSP é encontrada em www.ipd.uka.de/PSP/.

Quais atividades metodológicas são utilizadas durante o PSP?

²¹ Vale notar que os defensores do desenvolvimento ágil de software (Capítulo 3) também afirmam que o processo deve ficar próximo à equipe. Eles propõem um método alternativo para conseguir isso.

PONTO-CHAVE

O PSP enfatiza a necessidade de registrar e analisar tipos de erros cometidos, para que se possa elaborar estratégias para eliminá-los.

Autópsia. Usando as medidas e métricas coletadas (trata-se de um volume de dados substancial que deve ser analisado estatisticamente), é determinada a eficácia do processo. Medidas e métricas devem guiar as mudanças no processo de modo a melhorar sua eficiência.

O PSP enfatiza a necessidade de identificar erros precocemente e, tão importante quanto, compreender os tipos de erros que provavelmente ocorrerão. Isso é obtido por meio de uma rigorosa atividade de avaliação em todos os artefatos de software gerados.

O PSP representa uma abordagem disciplinada e baseada em métricas para a engenharia de software que pode causar um choque cultural em muitos profissionais. Entretanto, quando apresentado de forma apropriada aos engenheiros de software [Hum96], a melhoria resultante na produtividade da engenharia e na qualidade de software é significativa [Fer97]. Apesar disso, não foi adotado largamente pelo setor. Os motivos, infelizmente, têm mais a ver com a natureza humana e com a inércia organizacional do que com os pontos fortes e fracos da abordagem PSP. Esse processo é intelectualmente desafiador e exige um nível de comprometimento (por parte dos profissionais e de seus gerentes) que nem sempre é possível alcançar. O período de treinamento é relativamente longo e os custos de treinamento são altos. O nível de medição exigido é culturalmente difícil para muitos profissionais da área de software.

O PSP pode ser utilizado como um processo de software eficaz no nível pessoal? A resposta é um inequívoco “sim”. Porém, mesmo se não adotado em sua totalidade, muitos dos conceitos de aperfeiçoamento do processo pessoal que introduz são importantes e vale a pena aprendê-los.

2.6.2 Processo de Software em Equipe (TSP)

WebRef

Informações sobre a formação de equipes com alto desempenho empregando-se TSP e PSP podem ser obtidas em: www.sei.cmu.edu/tsp/.

Pelo fato de muitos projetos de software para nível industrial serem tratados por uma equipe de profissionais, Watts Humphrey estendeu as lições aprendidas com a introdução do PSP e propôs um *Processo de Software em Equipe (TSP, Team Software Process)*. O objetivo do TSP é criar uma equipe de projetos “autodirigida”, que se organize por si mesma para produzir software de alta qualidade. Humphrey [Hum98] define os seguintes objetivos para o TSP:

- Criar equipes autodirigidas que planejem e acompanhem seu próprio trabalho, estabeleçam metas e sejam proprietárias de seus processos e planos. As equipes poderão ser puras ou equipes de produto integradas (IPTs, integrated product teams) com cerca de 3 a 20 engenheiros.
- Mostrar aos gerentes como treinar e motivar suas equipes e como ajudá-las a manter alto desempenho.
- **Acelerar o aperfeiçoamento dos processos de software, tornando o comportamento CMM²² Nível 5 algo normal e esperado.**
- Fornecer orientação para melhorias a organizações com elevado grau de maturidade.
- Facilitar o ensino universitário de habilidades de trabalho em equipe de nível industrial.

Uma equipe autodirigida possui um entendimento consistente de suas metas e objetivos globais; define papéis e responsabilidades para cada um dos membros; monitora dados quantitativos de projeto (produtividade e qualidade); identifica um processo de equipe que seja apropriado para o projeto em questão e uma estratégia para implementação do processo; define padrões locais que sejam aplicáveis ao trabalho de engenharia da equipe; avalia continuamente os riscos e reage a eles e, finalmente, acompanha, gerencia e gera relatórios sobre a situação do projeto.

O TSP define as seguintes atividades metodológicas: **lançamento do projeto, projeto de alto nível, implementação, integração e testes e autópsia**. Assim como seus equivalentes no PSP (note que a terminologia é ligeiramente diferente), essas atividades capacitam a equipe a planejar, projetar e construir software de maneira disciplinada, ao mesmo tempo em



Para formar uma equipe autodirigida, deve haver boa colaboração internamente e boa comunicação externamente.

22 O Modelo de Maturidade de Capacidade (CMM, Capability Maturity Model), uma medida da eficiência de um processo de software, é discutido no Capítulo 30.

que mede quantitativamente o processo e o produto. A autópsia representa o estágio para melhorias dos processos.

Esse processo faz uso de uma grande variedade de roteiros (scripts), formulários e padrões que servem para orientar os membros da equipe em seu trabalho. Os roteiros definem atividades de processos específicas (isto é, lançamento do projeto, projeto, implementação, integração e testes do sistema, autópsia) e outras funções de trabalho mais detalhadas (por exemplo, planejamento do desenvolvimento, desenvolvimento de requisitos, gerenciamento das configurações de software, teste de unidade) que fazem parte do processo de equipe.

O TSP reconhece que as melhores equipes de software são autodirigidas.²³ Seus membros estabelecem os objetivos do projeto, adaptam o processo para atender suas necessidades, controlam o cronograma e, através de medições e análise das métricas coletadas, trabalham continuamente para aperfeiçoar a abordagem em relação à engenharia de software.

Assim como o PSP, o TSP é uma rigorosa abordagem da engenharia de software que fornece benefícios distintos e quantificáveis para a produtividade e para a qualidade. A equipe deve se comprometer totalmente com o processo e deve passar por treinamento consciente para assegurar que a abordagem seja apropriadamente aplicada.

PONTO-CHAVE

Os roteiros (scripts) do TSP definem os elementos e as atividades realizadas no transcorrer do processo.

2.7 TECNOLOGIA DE PROCESSOS

Um ou mais dos modelos de processo discutidos nas seções anteriores devem ser adaptados para ser empregados por uma equipe de software. Para tanto, desenvolveram-se *ferramentas de tecnologia de processos*, com o objetivo de auxiliar organizações de software a analisar seus processos atuais, organizar tarefas de trabalho, controlar e monitorar o progresso, bem como administrar a qualidade técnica.²⁴

As ferramentas de tecnologia de processos permitem a uma organização de software construir um modelo automatizado da metodologia de processos, conjuntos de tarefas e atividades de apoio (*umbrella activities*), discutidos na Seção 2.1. O modelo, normalmente representado como uma rede, pode, então, ser analisado para determinar o fluxo de trabalho típico e exa-

FERRAMENTAS DO SOFTWARE



Ferramentas de modelagem de processos

Objetivo: quando uma organização trabalha para aprimorar um processo de negócio (ou de software), ela precisa, primeiramente, compreendê-lo. As ferramentas de modelagem de processos (também chamadas ferramentas de tecnologia de processos ou ferramentas de gerenciamento de processos) são usadas para representar elementos-chave de um processo para que possa ser mais bem compreendido. Essas ferramentas podem também oferecer “links” para descrições de processos, ajudando os envolvidos no processo a compreender as ações e tarefas necessárias para realizá-lo. As ferramentas de modelagem de processos fornecem links para outras ferramentas que oferecem suporte para atividades de processos definidas.

Mecânica: as ferramentas nesta categoria permitem a uma equipe de desenvolvimento definir os elementos de um modelo

único de processo (ações, tarefas, artefato, pontos de garantia da qualidade de software), dar orientação detalhada sobre o conteúdo ou descrição de cada elemento de um processo e, então, gerenciar o processo conforme ele for conduzido. Em alguns casos, as ferramentas de tecnologia de processos incorporam tarefas padronizadas de gerenciamento de projeto como estimativa de custos, cronograma, acompanhamento e controle.

Ferramentas Representativas:

Igrafx Process Tools — ferramentas que capacitam uma equipe a mapear, medir e modelar o processo de software (www.micrografx.com)

Adeptia BPM Server — projetada para gerenciar, automatizar e otimizar processos de negócio (www.adeptia.com)

SpeedDev Suite — conjunto de seis ferramentas com forte ênfase no gerenciamento das atividades de comunicação e modelagem (www.speeddev.com)

²³ No Capítulo 3 discutiremos a importância das equipes “auto-organizadas” como um elemento-chave no desenvolvimento de software ágil.

²⁴ As ferramentas aqui citadas não representam um aval, mas sim uma amostragem de ferramentas nesta categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas de seus respectivos desenvolvedores.

minar estruturas de processos alternativas que possam levar à redução de custos e tempo de desenvolvimento.

Uma vez criado um processo aceitável, outras ferramentas de tecnologia de processo poderão ser usadas para alocar, monitorar e até mesmo controlar todas as atividades, ações e tarefas de engenharia de software definidas como parte do modelo de processo. Cada membro da equipe poderá usar tais ferramentas para desenvolver uma lista de controle das tarefas a ser realizadas, dos artefatos de software a ser gerados e das atividades de garantia da qualidade a ser realizadas. A ferramenta de tecnologia de processos também pode ser usada para coordenar o uso de outras ferramentas de engenharia de software que são apropriadas para uma determinada tarefa.

2.8 PROCESSO DO PRODUTO

Se o processo for fraco, certamente o produto final sofrerá consequências. Porém, uma confiança excessiva e obsessiva no processo é igualmente perigosa. Em um breve artigo, escrito muitos anos atrás, Margaret Davis [Dav95a] tece comentários atemporais sobre a dualidade produto e processo:

Aproximadamente a cada dez anos (acrescente ou elimine cinco), a comunidade de software redefine “o problema”, mudando seu foco de itens do produto para itens de processo. Assim, adotamos linguagens de programação estruturada (produto), seguidas por métodos de análise estruturada (processo), seguidos pelo encapsulamento de dados (produto), seguido pela ênfase atual no Modelo de Maturação da Capacidade de Desenvolvimento de Software (processo), do Software Engineering Institute [seguido por métodos orientados a objeto, seguido pelo desenvolvimento de software ágil].

Enquanto a tendência natural de um pêndulo é a de vir repousar num ponto intermediário entre dois extremos, o foco da comunidade de software muda constantemente, pois nova força é aplicada quando a última oscilação falha. Essas oscilações causam danos para si mesmos e para o ambiente externo, confundindo o profissional típico de software, mudando radicalmente o que significava desempenhar bem seu trabalho. Essas oscilações também não resolvem “o problema”, pois estão fadadas ao insucesso, enquanto produto e processo forem tratados como formando uma dicotomia (divisão de um conceito em dois elementos, em geral, contrários) em vez de uma dualidade (coexistência de dois princípios).

Na comunidade científica, há precedentes da tendência para adotar noções de dualidade quando, nas observações, as contradições não podem ser explicadas completamente nem por uma nem por outra teoria que competem entre si. A natureza dual da luz, parecendo ser simultaneamente partícula e onda, foi aceita desde os anos 1920, quando Louis de Broglie a propôs. Pelas observações feitas dos artefatos de software e de seu desenvolvimento, fica demonstrada a existência de uma dualidade fundamental entre produto e processo. Jamais poderemos destrinchar ou compreender o artefato completo, seu contexto, uso, significado e valor se o enxergarmos apenas ou como um processo ou um produto...

Todas as atividades humanas podem ser um processo, mas todos sentem-se valorizados quando tais atividades se tornam uma representação ou um exemplo, sendo utilizadas ou apreciadas por mais de uma pessoa, repetidamente, ou então utilizadas num contexto não imaginado. Ou seja, extraímos sentimentos de satisfação na reutilização de nossos produtos, seja por nós mesmos, seja por outros.

Assim, enquanto a assimilação rápida das metas de reúso, no desenvolvimento de software, aumenta potencialmente a satisfação dos profissionais de software, ela também aumenta a urgência da aceitação da dualidade produto e processo. Enxergar um artefato reutilizável apenas como um produto ou apenas como um processo, obscurece o contexto e as maneiras de usá-lo, ou obscurece o fato de que cada uso resulta em produto que, por sua vez, será utilizado como entrada, em alguma outra atividade de desenvolvimento de software. Adotar

uma dessas visões em detrimento da outra reduz dramaticamente as oportunidades de reutilização e, portanto, perde-se a oportunidade de aumentar a satisfação no trabalho.

As pessoas obtêm satisfação tanto do processo criativo quanto do produto final. Um artista sente prazer tanto de suas pinceladas quanto do resultado geral de seu quadro. Um escritor sente prazer tanto da procura da metáfora apropriada quanto do livro finalizado. Como profissional de software criativo, você também deve extrair tanta satisfação do processo como do produto final. A dualidade produto e processo é um elemento importante para manter pessoas criativas engajadas à medida que a engenharia de software continua a evoluir.

2.9 RESUMO

Um modelo de processo genérico para engenharia de software consiste num conjunto de atividades metodológicas e de apoio (*umbrella activities*), ações e tarefas a realizar. Cada modelo de processo, dentre os vários existentes, pode ser descrito por um fluxo de processo diferente — descrição de como as atividades metodológicas, ações e tarefas são organizadas sequencial e cronologicamente. Padrões de processo são utilizados para resolver problemas comuns encontrados como parte do processo de software.

Os modelos de processo prescritivos são aplicados há anos, num esforço para organizar e estruturar o desenvolvimento de software. Cada um desses modelos sugere um fluxo de processos ligeiramente diferente, mas todos realizam o mesmo conjunto de atividades metodológicas genéricas: comunicação, planejamento, modelagem, construção e emprego.

Os modelos de processo sequenciais, tais como o de cascata e o modelo V, são os paradigmas da engenharia de software mais antigos. Eles sugerem um fluxo de processos linear que, frequentemente, é inadequado para considerar as características dos sistemas modernos (por exemplo, contínuas alterações, sistemas em evolução, prazos apertados). Entretanto, eles têm, realmente, aplicabilidade em situações em que os requisitos são bem definidos e estáveis.

Modelos de processo incremental são iterativos por natureza e produzem rapidamente versões operacionais do software. Modelos de processos evolucionários reconhecem a natureza iterativa e incremental da maioria dos projetos de engenharia de software e são projetados para adequar mudanças. Esses modelos, como prototipação e o modelo espiral, produzem rapidamente artefatos de software incrementais (ou versões operacionais do software). Podem ser adotados para ser aplicados por todas as atividades de engenharia de software — desde o desenvolvimento de conceitos até a manutenção do sistema a longo prazo.

Modelo de processo concorrente possibilita que uma equipe de software represente elementos iterativos e concorrentes de qualquer modelo de processo. Modelos especializados incluem o modelo baseado em componentes (que enfatiza a montagem e a reutilização de componentes); o modelo de métodos formais (que encoraja uma abordagem matemática para o desenvolvimento e a verificação de software); e o modelo orientado a aspectos (que considera interesses cruzados que se estendem por toda a arquitetura do sistema). O Processo Unificado é um processo de software “dirigido a casos práticos, centrado na arquitetura, iterativo e incremental”, desenvolvido como uma metodologia para os métodos e ferramentas da UML.

Modelos pessoal e de equipe enfatizam a medição, o planejamento e autodirecionamento como ingredientes-chave para um processo de software bem-sucedido.

PROBLEMAS E PONTOS A PONDERAR

2.1. Na introdução deste capítulo, Baetjer observa: “O processo oferece interação entre usuários e projetistas, entre usuários e ferramentas em evolução e entre projetistas e ferramentas [de tecnologia] em evolução”. Liste cinco perguntas que (a) os projetistas deveriam fazer aos usuários, (b) os usuários deveriam fazer aos projetistas, (c) os usuários deveriam fazer a si mesmos sobre o produto de software a ser desenvolvido, (d) os projetistas deveriam fazer

a si mesmos sobre o produto de software a ser construído e sobre o processo que será usado para construí-lo.

2.2. Tente desenvolver um conjunto de ações para a atividade de comunicação. Selecione uma ação e defina um conjunto de tarefas para ela.

2.3. Durante a **comunicação, um problema comum ocorre** ao encontrarmos dois interessados com ideias conflitantes sobre como o software deveria ser. Isto é, há requisitos mutuamente conflitantes. Desenvolva um padrão de processo (que seja um padrão de estágio) usando o modelo apresentado na Seção 2.1.3 que se refere a esse problema, e sugira uma abordagem efetiva para ele.

2.4. Pesquise sobre o PSP e faça uma breve apresentação descrevendo os tipos de medidas que um engenheiro de software individual deve fazer e como tais medidas podem ser usadas para aprimorar sua eficácia pessoal.

2.5. O uso de roteiros (scripts — um mecanismo exigido no TSP) não é universalmente apreciado na comunidade de software. Faça uma lista dos prós e contras referentes aos roteiros e sugira pelo menos duas situações nas quais seriam úteis e outras duas onde poderiam oferecer menos benefício.

2.6. Leia [Nog00] e redija um artigo de duas ou três páginas que discuta o impacto do “caos” na engenharia de software.

2.7. Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo casca-ta. Seja específico.

2.8. Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo de prototipação. Seja específico.

2.9. Quais adaptações de processo seriam necessárias caso o protótipo fosse se transformar em um sistema ou produto a ser entregue?

2.10. Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo incremental. Seja específico.

2.11. À medida que se desloca para fora, ao longo do fluxo de processo em espiral, o que pode ser dito em relação ao software que está sendo desenvolvido ou sofrendo manutenção?

2.12. É possível combinar modelos de processo? Em caso positivo, dê um exemplo.

2.13. O modelo de processo concorrente define um conjunto de “estados”. Descreva, com suas próprias palavras, o que esses estados representam e, em seguida, indique como entram em cena no modelo de processos concorrentes.

2.14. Quais são as vantagens e desvantagens em desenvolver software cuja qualidade é “boa o suficiente”? Ou seja, o que acontece quando enfatizamos a velocidade de desenvolvimento em detrimento da qualidade do produto?

2.15. Forneça três exemplos de projetos de software que seriam suscetíveis ao modelo baseado em componentes. Seja específico.

2.16. É possível provar que um componente de software e até mesmo um programa inteiro está correto. Então, por que todo mundo não faz isso?

2.17. Processo Unificado e UML são a mesma coisa? Justifique sua resposta.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A maioria dos livros texto sobre engenharia de software considera os modelos de processo tradicionais com certo nível de detalhe. Livros como os de Sommerville (*Software Engineering*, 8. ed., Addison-Wesley, 2006), Pfleeger e Atlee (*Software Engineering*, 3. ed., Prentice-Hall, 2005) e Schach (*Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*, 7. ed., McGraw-Hill, 2009) falam sobre os paradigmas tradicionais e discutem seus pontos fortes e fracos. Glass (*Facts and Fallacies of Software Engineering*, Prentice-Hall, 2002) fornece

uma visão nua e crua, bem como pragmática, do processo de engenharia de software. Embora não seja especificamente dedicado a processos, Brooks (*The Mythical Man-Month*, 2. ed., Addison-Wesley, 1995) apresenta conhecimentos sábios de projeto antigo que têm tudo a ver com processos.

Firesmith e Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) apresenta um quadro geral para a criação de “processos de software flexíveis e que, ainda assim, não deixam de ser disciplinados” e discute atributos e objetivos dos processos. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) fala sobre técnicas de modelagem que possibilitam a análise dos elementos técnicos e sociais inter-relacionados do processo de software. Sharpe e McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, Artech House, 2001) apresenta ferramentas para modelagem de processos de software, bem como de processos de negócios.

Lim (*Managing Software Reuse*, Prentice Hall, 2004) discute a reutilização sob a perspectiva gerencial. Ezran, Morisio e Tully (*Practical Software Reuse*, 2002) e Jacobson, Griss e Jonsson (*Software Reuse*, Addison-Wesley, 1997) apresentam informações muito úteis sobre desenvolvimento baseado em componentes. Heineman e Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) descrevem o processo exigido para implementar sistemas baseados em componentes. Kenett e Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) colocam como a gestão da qualidade e o projeto de processos estão intimamente ligados entre si.

Nygard (*Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007) e Richardson e Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) apresentam um amplo conjunto de diretrizes úteis aplicáveis à atividade de emprego.

Além do livro seminal de Jacobson, Rumbaugh e Booch sobre o Processo Unificado [Jac99], livros como os de Arlow e Neustadt (*UML 2 and the Unified Process*, Addison-Wesley, 2005), Kroll e Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003) e Farve (*UML and the Unified Process*, IRM Press, 2003) fornecem excelentes informações complementares. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) fala sobre o gerenciamento de projetos no contexto do PU.

Uma ampla variedade de fontes de informação sobre engenharia de software e o processo de software está disponível na Internet. Uma lista atualizada de referências relevantes para o processo de software pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

DESENVOLVIMENTO ÁGIL

CONCEITOS-
-CHAVE

agilidade	.82
Crystal.....	.97
desenvolvimento de software adaptativo	.94
desenvolvimento de software enxuto (LSD)	.99
DSDM	.96
Extreme Programming – XP (programação extrema)	.87
FDD	.98
histórias.....	.89
processo ágil.....	.85
processo unificado ágil	.101
processo XP	.89
programação em duplas	.88

Em 2001, Kent Beck e outros dezesseis renomados desenvolvedores, autores e consultores da área de software [Bec01a] (batizados de “Agile Alliance”- “Aliança dos Ágeis”) assinaram o “Manifesto para o Desenvolvimento Ágil de Software” (“Manifesto for Agile Software Development”), que se inicia da seguinte maneira:

Desenvolvendo e ajudando outros a desenvolver software, estamos desvendando formas melhores de desenvolvimento. Por meio deste trabalho passamos a valorizar:

Indivíduos e interações acima de processos e ferramentas

Software operacional acima de documentação completa

Colaboração dos clientes acima de negociação contratual

Respostas a mudanças acima de seguir um plano

Ou seja, embora haja valor nos itens à direita, valorizaremos os da esquerda mais ainda.

Normalmente, um manifesto é associado a um movimento político emergente: atacando a velha guarda e sugerindo uma mudança revolucionária (espera-se que para melhor). De certa forma, é exatamente do que trata o desenvolvimento ágil.

Embora as ideias básicas que norteiam o desenvolvimento ágil tenham estado conosco por muitos anos, só há menos de duas décadas que se consolidaram como um “movimento”.

PANORAMA

O que é? A engenharia de software ágil combina filosofia com um conjunto de princípios de desenvolvimento. A filosofia defende a satisfação do cliente e a entrega de incremental prévio; equipes de projeto pequenas e altamente motivadas; métodos informais; artefato de engenharia de software mínimos e, acima de tudo, simplicidade no desenvolvimento geral. Os princípios de desenvolvimento priorizam a entrega mais que análise e projeto (embora essas atividades não sejam desencorajadas); também priorizam a comunicação ativa e contínua entre desenvolvedores e clientes.

Quem realiza? Os engenheiros de software e outros envolvidos no projeto (gerentes, clientes, usuários finais) trabalham conjuntamente em uma equipe ágil — uma equipe que se auto-organiza e que controla seu próprio destino. Uma equipe ágil acelera a comunicação e a colaboração entre todos os participantes (que estão a seu serviço).

Por que é importante? O moderno ambiente dos sistemas e dos produtos da área é acelerado e está em constante mudança. A engenharia de software ágil constitui uma razoável alternativa para a engenharia convencional voltada para certas

classes de software e para certos tipos de projetos, e tem se mostrado capaz de entregar sistemas corretos rapidamente.

Quais são as etapas envolvidas? O desenvolvimento ágil poderia ser mais bem denominado “engenharia de software flexível”. As atividades metodológicas básicas permanecem: comunicação, planejamento, modelagem, construção e emprego. Entretanto, estas se transformam em um conjunto de tarefas mínimas que impulsiona a equipe para o desenvolvimento e para a entrega (pode-se levantar a questão de que isso é feito em detrimento da análise do problema e do projeto de soluções).

Qual é o artefato? Tanto o cliente como o engenheiro têm o mesmo parecer: o único artefato realmente importante consiste em um “incremento de software” operacional que seja entregue, adequadamente, na data combinada.

Como garantir que o trabalho foi realizado corretamente? Se a equipe ágil concordar que o processo funciona e essa equipe produz incrementos de software passíveis de entrega e que satisfaçam o cliente, então, o trabalho está correto.

retrabalho	87
scrum	95
velocidade	
de projeto.	86
XP Industrial. . .	91

Em essência, métodos ágeis¹ se desenvolveram em um esforço para sanar fraquezas reais e perceptíveis da engenharia de software convencional. O desenvolvimento ágil oferece benefícios importantes, no entanto, não é indicado para todos os projetos, produtos, pessoas e situações. Também não é a antítese da prática de engenharia de software consistente e pode ser aplicado como uma filosofia geral para todos os trabalhos de software.

Na economia moderna é frequentemente difícil ou impossível prever como um sistema computacional (por exemplo, uma aplicação baseada na Web) irá evoluir com o tempo. As condições de mercado mudam rapidamente, as necessidades dos usuários finais se alteram e novas ameaças competitivas emergem sem aviso. Em muitas situações, não se conseguirá definir completamente requisitos antes que se inicie o projeto. É preciso ser ágil o suficiente para dar uma resposta ao ambiente de fluido negócios.

Fluidez implica mudanças, e mudanças são caras. Particularmente, se forem sem controle e mal gerenciadas. Uma das características mais convincentes da abordagem ágil é sua habilidade de reduzir os custos da mudança ao longo de todo o processo de software.

Isso significa que o reconhecimento dos desafios apresentados pela moderna realidade faz com que se descartem valiosos princípios da engenharia de software, conceitos, métodos e ferramentas? Absolutamente não! Como todas as disciplinas de engenharia, a engenharia de software continua a evoluir, podendo ser adaptada facilmente aos desafios apresentados pela demanda por agilidade.

Em um texto que nos leva à reflexão sobre desenvolvimento de software ágil, Alistair Cockburn [Coc02] argumenta que o modelo de processo prescritivo, apresentado no Capítulo 2, tem uma falha essencial: esquece das fragilidades das pessoas que desenvolvem o software. Os engenheiros de software não são robôs. Eles apresentam grande variação nos estilos de trabalho; diferenças significativas no nível de habilidade, criatividade, organização, consistência e espontaneidade. Alguns se comunicam bem na forma escrita, outros não. Cockburn afirma que os modelos de processos podem “lidar com as fraquezas comuns das pessoas com disciplina e/ou tolerância” e que a maioria dos modelos de processos prescritivos opta por disciplina. Segundo ele: “Como a consistência nas ações é uma fraqueza humana, as metodologias com disciplina elevada são frágeis”.

Para que funcionem, os modelos de processos devem fornecer um mecanismo realista que estimule a disciplina necessária ou, então, devem ter características que apresentem “tolerância” com as pessoas que realizam trabalhos de engenharia de software. Invariavelmente, práticas tolerantes são mais facilmente adotadas e sustentadas pelas pessoas envolvidas, porém (como o próprio Cockburn admite) podem ser menos produtivas. Como a maioria das coisas na vida, deve-se considerar os prós e os contras.

“Agilidade: 1,
todo o resto: 0.”

Tom DeMarco

3.1 O QUE É AGILIDADE?

No contexto da engenharia de software, o que é agilidade? Ivar Jacobson [Jac02a] apresenta uma útil discussão:

Atualmente, *agilidade* tornou-se a palavra da moda quando se descreve um moderno processo de software. Todo mundo é ágil. Uma equipe ágil é aquela rápida e capaz de responder apropriadamente a mudanças. Mudanças têm muito a ver com desenvolvimento de software. Mudanças no software que está sendo criado, mudanças nos membros da equipe, mudanças devido a novas tecnologias, mudanças de todos os tipos que poderão ter um impacto no produto que está em construção ou no projeto que cria o produto. Suporte para mudanças deve ser incorporado em tudo o que fazemos em software, algo que abraçamos porque é o coração e a alma do software. Uma equipe ágil reconhece que o software é desenvolvido por indivíduos trabalhando em equipes e que as habilidades dessas pessoas, suas capacidades em colaborar estão no cerne do sucesso do projeto.

1 Os métodos ágeis são algumas vezes conhecidos como *métodos light* ou *métodos enxutos* (*lean methods*).



Não cometa o erro de assumir que a agilidade lhe dará licença para abreviar soluções. Processo é um requisito e disciplina é essencial.

Segundo Jacobson, a penetração da mudança é o principal condutor para a agilidade. Os engenheiros de software devem ser rápidos em seus passos caso queiram assimilar as rápidas mudanças que Jacobson descreve.

Porém, agilidade consiste em algo mais que uma resposta à mudança, abrangendo a filosofia proposta no manifesto citado no início deste capítulo. Ela incentiva a estruturação e as atitudes em equipe que tornam a comunicação mais fácil (entre membros da equipe, entre o pessoal ligado à tecnologia e o pessoal da área comercial, entre os engenheiros de software e seus gerentes). Enfatiza a entrega rápida do software operacional e diminui a importância dos artefatos intermediários (nem sempre um bom negócio); assume o cliente como parte da equipe de desenvolvimento e trabalha para eliminar a atitude de “nós e eles”, que continua a invadir muitos projetos de software; reconhece que o planejamento em um mundo incerto tem seus limites e que o plano (roteiro) de projeto deve ser flexível.

A agilidade pode ser aplicada a qualquer processo de software. Entretanto, para obtê-la, é essencial que seja projetado para que a equipe possa adaptar e alinhar (racionalizar) tarefas; possa conduzir o planejamento compreendendo a fluidez de uma abordagem do desenvolvimento ágil; possa eliminar tudo, exceto os artefatos essenciais, conservando-os enxutos; e enfatize a estratégia de entrega incremental, conseguindo entregar ao cliente, o mais rapidamente possível, o software operacional para o tipo de produto e ambiente operacional.

3.2 AGILIDADE E O CUSTO DAS MUDANÇAS

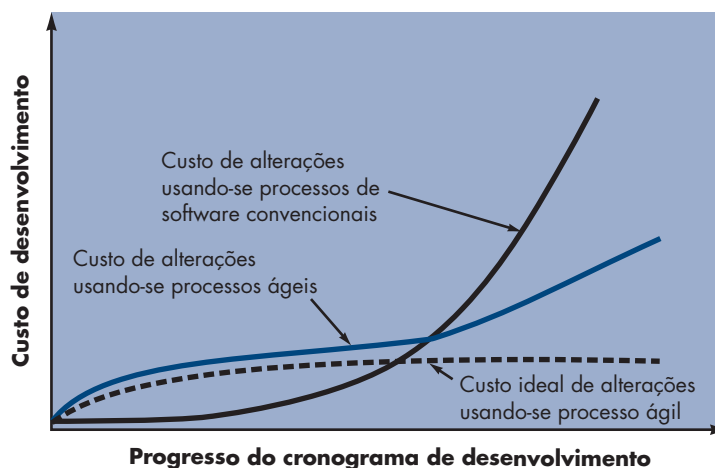
“A agilidade é dinâmica, de conteúdo específico, abrange mudanças agressivas e é orientada ao crescimento.”

Steven Goldman et al.

A sabedoria convencional em desenvolvimento de software (baseada em décadas de experiência) afirma que os custos de mudanças aumentam de forma não linear conforme o projeto avança (Figura 3.1, curva em preto contínuo). É relativamente fácil assimilar uma mudança quando uma equipe de software está juntando requisitos (no início de um projeto). Pode-se ter de alterar um detalhamento do uso, ampliar uma lista de funções ou editar uma especificação por escrito. Os custos de tal trabalho são mínimos e o tempo demandado não afetará negativamente o resultado do projeto. Mas se adiarmos alguns meses, o que aconteceria? A equipe está em meio aos testes de validação (que ocorre relativamente no final do projeto) e um importante interessado está requisitando uma mudança funcional de vulto. A mudança requer uma alteração no projeto da arquitetura do software, o projeto e desenvolvimento de três novos componentes, modificações em outros cinco componentes, projeto de novos testes, e assim por diante. Os custos crescem rapidamente e não serão triviais o tempo e custos necessários para assegurar que a mudança seja feita sem efeitos colaterais inesperados.

FIGURA 3.1

Custos de alterações como uma função do tempo em desenvolvimento



PONTO-CHAVE

Um processo ágil reduz o custo das alterações porque o software é entregue (liberado) de forma incremental e as alterações podem ser mais bem controladas dentro de incrementais.

Os defensores da agilidade (por exemplo, [Bec00], [Amb04]) argumentam que um processo ágil bem elaborado “achata” o custo da curva de mudança (Figura 3.1, curva em linha verde), permitindo que uma equipe de software assimile as alterações, realizadas posteriormente em um projeto de software, sem um impacto significativo nos custos ou no tempo. Já foi mencionado que o processo ágil envolve entregas incrementais. O custo das mudanças é atenuado quando a entrega incremental é associada a outras práticas ágeis, como testes contínuos de unidades e programação por pares, (discutida adiante neste capítulo). Há evidências [Coc01a] que sugerem que se pode alcançar redução significativa nos custos de alterações, embora haja um debate contínuo sobre qual o nível em que a curva de custos torna-se “achatada”.

3.3 O QUE É PROCESSO ÁGIL?

Qualquer processo ágil de software é caracterizado de uma forma que se relacione a uma série de preceitos-chave [Fow02] acerca da maioria dos projetos de software:

1. É difícil afirmar antecipadamente quais requisitos de software irão persistir e quais sofrerão alterações. É igualmente difícil prever de que maneira as prioridades do cliente sofrerão alterações conforme o projeto avança.
2. Para muitos tipos de software, o projeto e a construção são “interconduzidos”. Ou seja, ambas as atividades devem ser realizadas em sequência (uma atrás da outra), para que os modelos de projeto sejam provados conforme sejam criados. É difícil prever quanto de trabalho de projeto será necessário antes que a sua construção (desenvolvimento) seja implementada para avaliar o projeto.
3. Análise, projeto, construção (desenvolvimento) e testes não são tão previsíveis (do ponto de vista de planejamento) quanto gostaríamos que fosse.

Dados esses três preceitos, surge uma importante questão: Como criar um processo capaz de administrar a *imprevisibilidade*? A resposta, conforme já observado, consiste na adaptabilidade de processo (para alterar rapidamente o projeto e as condições técnicas). Portanto, um processo ágil deve ser *adaptável*.

Mas adaptação contínua sem progressos que levem em frente o desenvolvimento realiza muito pouco. Um processo ágil de software deve adaptar *incrementalmente*. Para conseguir uma adaptação incremental, a equipe ágil precisa de feedback do cliente (de modo que as adaptações apropriadas possam ser feitas). Um efetivo catalisador para feedback de cliente é um protótipo operacional ou parte de um sistema operacional. Dessa forma, deve se instituir uma *estratégia de desenvolvimento incremental*. Os *incrementos de software* (protótipos executáveis ou partes de um sistema operacional) devem ser entregues em curtos períodos de tempo, de modo que as adaptações acompanhem o mesmo ritmo das mudanças (imprevisibilidade). Essa abordagem iterativa capacita o cliente a avaliar o incremento de software regularmente, fornecer o feedback necessário para a equipe de software e influenciar as adaptações de processo feitas para incluir adequadamente o feedback.

3.3.1 Princípios da agilidade

A Agile Alliance (veja [Agi03], [Fow01]) estabelece 12 princípios de agilidade para quem quiser ter agilidade:

1. A maior prioridade é satisfazer o cliente por meio de entrega adiantada e contínua de software valioso.
2. Acolha bem os pedidos de alterações, mesmo atrasados no desenvolvimento. Os processos ágeis se aproveitam das mudanças como uma vantagem competitiva na relação com o cliente.

WebRef

Uma vasta coleção de artigos sobre processo ágil pode ser encontrada em www.aanpo.org/articles/index.

PONTO-CHAVE

Embora processos ágeis considerem as alterações, examinar as razões para tais mudanças ainda continua sendo importante.



Software ativo é importante, mas não se deve esquecer que também deve apresentar uma série de atributos de qualidade, incluindo confiabilidade, usabilidade e facilidade de manutenção.

3. Entregue software em funcionamento frequentemente, de algumas semanas para alguns meses, dando preferência a intervalos mais curtos.
4. O pessoal comercial e os desenvolvedores devem trabalhar em conjunto diariamente ao longo de todo o projeto.
5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e apoio necessários e confie neles para ter o trabalho feito.
6. O método mais eficiente e efetivo de transmitir informações para e dentro de uma equipe de desenvolvimento é uma conversa aberta, de forma presencial
7. Software em funcionamento é a principal medida de progresso.
8. Os processos ágeis promovem desenvolvimento sustentável. Os proponentes, desenvolvedores e usuários devem estar capacitados para manter um ritmo constante indefinidamente.
9. Atenção contínua para com a excelência técnica e para com bons projetos aumenta a agilidade.
10. Simplicidade — a arte de maximizar o volume de trabalho não efetuado — é essencial.
11. As melhores arquiteturas, requisitos e projetos emergem de equipes que se auto-organizam.
12. A intervalos regulares, a equipe se avalia para ver como tornar-se mais eficiente, então sintoniza e ajusta seu comportamento de acordo.

Nem todo modelo de processo ágil aplica esses 12 princípios atribuindo-lhes pesos iguais, e alguns modelos preferem ignorar (ou pelo menos relevam) a importância de um ou mais desses princípios. Entretanto, os princípios definem um *espírito ágil* mantido em cada um dos modelos de processo apresentados neste capítulo.

3.3.2 A política do desenvolvimento ágil

Há debates consideráveis (algumas vezes acirrados) sobre os benefícios e a aplicabilidade do desenvolvimento de software ágil em contraposição a processos de engenharia de software mais convencionais. Jim Highsmith [Hig02a] (em tom jocoso) estabelece extremos ao caracterizar o sentimento do grupo pró-agilidade (“os agilistas”). “Os metodologistas tradicionais são um bando de ‘pés na lama’ que preferem produzir documentação sem falhas em vez de um sistema que funcione e atenda às necessidades do negócio.” Em um contraponto, ele apresenta (mais uma vez, em tom jocoso) a posição do grupo da engenharia de software tradicional: “Os metodologistas de pouco peso, quer dizer, os metodologistas ‘ágeis’ são um bando de hackers pretensiosos que vão acabar tendo uma grande surpresa ao tentarem transformar seus brinquedinhos em software de porte empresarial”.



Você não tem de escolher entre agilidade ou engenharia de software. Em vez disso, defina uma abordagem de engenharia de software que seja ágil.

Como toda argumentação sobre tecnologia de software, o debate sobre metodologia corre o risco de descambar para uma guerra santa. Se for deflagrada uma guerra, a racionalidade desaparece e crenças, em vez de fatos, orientarão a tomada de decisão.

Ninguém é contra a agilidade. A verdadeira questão é: Qual a melhor maneira de atingi-la? Igualmente importante, como desenvolver software que atenda às necessidades atuais dos clientes e que apresente características de qualidade que permitirão que seja estendido e ampliado para responder às necessidades dos clientes no longo prazo?

Não há respostas absolutas para nenhuma dessas questões. Mesmo na própria escola ágil, existem vários modelos de processos propostos (Seção 3.4), cada um com uma abordagem sutilmente diferente a respeito do problema da agilidade. Em cada modelo existe um conjunto de “ideias” (os agilistas relutam em chamá-las “tarefas de trabalho”) que representam um afastamento significativo da engenharia de software tradicional. E, ainda assim, muitos conceitos ágeis são apenas adaptações de bons conceitos da engenharia de software. Conclusão: pode-se ganhar muito considerando o que há de melhor nas duas escolas e praticamente nada denegrindo uma ou outra abordagem.

Caso se interesse mais, veja [Hig01], [Hig02a] e [DeM02], em que é apresentado um sumário interessante a respeito de outras questões técnicas e políticas importantes.

“Muito da agilidade dos métodos deriva do fato de terem suas bases no conhecimento tácito incorporado pela e na equipe, em vez de registrar por escrito tal conhecimento em planejamentos.”

Barry Boehm

? Quais são as características-chave que devem estar presentes entre as pessoas integrantes de uma equipe de software eficiente?

“O que é visto como razoavelmente suficiente por uma equipe pode ser avaliado como mais do que suficiente ou insuficiente por uma outra equipe.”

Alistair Cockburn

PONTO-CHAVE

Uma equipe auto-organizada está no controle do trabalho que realiza. A equipe estabelece seus próprios compromissos e define planos para cumprí-los.

3.3.3 Fatores humanos

Os defensores do desenvolvimento de software ágil se esmeram para enfatizar a importância dos “fatores humanos”. Como afirmam Cockburn e Highsmith [Coc01a], “O desenvolvimento ágil foca talentos e habilidades de indivíduos, moldando o processo de acordo com as pessoas e as equipes específicas”. O ponto-chave nessa afirmação é que *o processo se amolda às necessidades das pessoas e equipes*, e não o caminho inverso.²

Se os membros da equipe de software devem orientar as características do processo que é aplicado para construir software, deve existir um certo número de traços-chave entre as pessoas de uma equipe ágil e a equipe em si:

Competência. No contexto do desenvolvimento ágil (assim como no da engenharia de software), a “competência” abrange talento inato, habilidades específicas relacionadas a software e conhecimento generalizado do processo que a equipe escolheu para aplicar. Habilidade e conhecimento de processo podem e devem ser ensinados para todas as pessoas que sejam membros de uma equipe ágil.

Foco comum. Embora os membros de uma equipe ágil possam realizar diferentes tarefas e tragam diferentes habilidades para o projeto, todos devem estar focados em um único objetivo — entregar um incremento de software funcionando ao cliente, dentro do prazo prometido. Para alcançar essa meta, a equipe também irá focar em adaptações contínuas (pequenas e grandes) que farão com que o processo se ajuste às necessidades da equipe.

Colaboração. Engenharia de software (independentemente do processo) trata de avaliação, análise e uso de informações comunicadas à equipe de software; criar informações que ajudarão todos os envolvidos a compreender o trabalho da equipe e a construir informações (software para computadores e bancos de dados relevantes) que forneçam valor de negócio para o cliente. Para realizar essas tarefas, os membros da equipe devem colaborar — entre si e com todos os demais envolvidos.

Habilidade na tomada de decisão. Qualquer boa equipe de software (até mesmo as equipes ágeis) deve ter liberdade para controlar seu próprio destino. Isso implica que seja dada autonomia à equipe — autoridade na tomada de decisão, tanto em assuntos técnicos como de projeto.

Habilidade de solução de problemas confusos. Os gerentes de software devem reconhecer que a equipe ágil terá de lidar continuamente com a ambiguidade e que será continuamente atingida por mudanças. Em alguns casos, a equipe tem de aceitar o fato de que o problema que eles estão solucionando hoje talvez não seja o problema que necessita ser solucionado amanhã. Entretanto, lições aprendidas de qualquer atividade de solução de problemas (inclusive aquelas que resolvem o problema errado) podem ser, futuramente, benéficas para a equipe no projeto.

Confiança mútua e respeito. A equipe ágil deve tornar-se uma equipe tal qual a que DeMarco e Lister [DeM98] denominam de equipe “consistente” (Capítulo 24). Uma equipe consistente demonstra a confiança e o respeito necessários para torná-la “tão fortemente unida que o todo fica maior do que a soma das partes”. [DeM98]

Auto-organização. No contexto do desenvolvimento ágil, a auto-organização implica três fatores: (1) a equipe ágil se organiza para o trabalho a ser feito, (2) a equipe organiza o processo para melhor se adequar ao seu ambiente local, (3) a equipe organiza o cronograma de trabalho para melhor cumprir a entrega do incremento de software. A auto-organização possui uma série de benefícios técnicos, porém, mais importante, é o fato de servir para melhorar a colaboração e levantar o moral da equipe. Em essência, a equipe faz seu próprio

2 As organizações de engenharia de software bem-sucedidas reconhecem essa realidade independentemente do modelo de processos por elas escolhido.

gerenciamento. Ken Schwaber [Sch02] menciona tais características ao escrever: “A equipe seleciona quanto trabalho acredita ser capaz de realizar dentro da iteração e se compromete com trabalho. Nada desmotiva tanto uma equipe como um terceiro assumir compromissos por ela. Nada motiva tanto uma equipe quanto aceitar a responsabilidade de cumprir completamente o prometido feito por ela própria”.

3.4 EXTREME PROGRAMMING – XP (PROGRAMAÇÃO EXTREMA)

Para ilustrar um processo ágil de forma um pouco mais detalhada, segue uma visão geral de *Extreme Programming – XP (Programação Extrema)*, a abordagem mais amplamente utilizada para desenvolvimento de software ágil. Embora os primeiros trabalhos sobre os conceitos e métodos associados à XP tenham ocorrido durante o final dos anos 1980, o trabalho seminal sobre o tema foi escrito por Kent Beck [Bec04a]. Mais recentemente, foi proposta uma variação da XP, denominada *Industrial XP (IXP)* [Ker05]. A IXP refina a XP e visa o processo ágil especificamente para uso em grandes organizações.

3.4.1 Valores da XP

Beck [Bec04a] define um conjunto de cinco *valores* que estabelecem as bases para todo trabalho realizado como parte da XP — comunicação, simplicidade, feedback (realimentação ou retorno), coragem e respeito. Cada um desses valores é usado como um direcionador das atividades, ações e tarefas específicas da XP.

Para conseguir a *comunicação* efetiva entre engenheiros de software e outros envolvidos (por exemplo, estabelecer os fatores e funções necessárias para o software), a XP enfatiza a colaboração estreita, embora informal (verbal), entre clientes e desenvolvedores, o estabelecimento de metáforas eficazes³ para comunicar conceitos importantes, feedback (realimentação) contínuo e evitar documentação volumosa como meio de comunicação.

Para alcançar a *simplicidade*, a XP restringe os desenvolvedores a projetar apenas para as necessidades imediatas, em vez de considerarem as necessidades futuras. O intuito é criar um projeto simples que possa ser facilmente implementado em código. Se o projeto tiver que ser melhorado, ele poderá ser *refabricado*⁴ mais tarde.

O *feedback* provém de três fontes: do próprio software implementado, do cliente e de outros membros da equipe de software. Através da elaboração do projeto e da implementação de uma estratégia de testes eficaz (Capítulos 17 a 20), o software (via resultados de testes) propicia um feedback para a equipe ágil. A XP faz uso do *teste de unidades* como sua tática de testes primária. À medida que cada classe é desenvolvida, a equipe desenvolve um teste de unidades para exercitar cada operação de acordo com sua funcionalidade especificada. À medida que um incremento é entregue a um cliente, as *histórias de usuários* ou *casos de uso* (Capítulo 5) implementados pelo incremento são usados como base para testes de aceitação. O grau em que o software implementa o produto, a função e o comportamento do caso em uso é uma forma de feedback. Por fim, conforme novas necessidades surgem como parte do planejamento iterativo, a equipe dá ao cliente um rápido feedback referente ao impacto nos custos e no cronograma.

Beck [Bec04a] afirma que a adoção estrita a certas práticas da XP exige *coragem*. Uma palavra melhor poderia ser *disciplina*. Por exemplo, frequentemente, há uma pressão significativa para a elaboração do projeto pensando em futuros requisitos. A maioria das equipes de software sucumbe, argumentando que “projetar para amanhã” poupará tempo e esforço no longo prazo. Uma equipe XP ágil deve ter disciplina (coragem) para projetar para hoje, reconhecendo que as



AVISO

Simplifique sempre que puder, mas tenha ciência de que um “retrabalho” (refabricação, redensenvolvimento) contínuo consegue absorver tempo e recursos significativos.

“A XP é a resposta para a pergunta: ‘Qual o mínimo possível que se pode realizar e mesmo assim desenvolver um software grandioso?’.”

Anônimo

³ No contexto da XP, uma *metáfora* é “uma história que todos — clientes, programadores e gerentes — podem contar sobre como o sistema funciona” [Bec04a].

⁴ A refabricação permite a um engenheiro de software aperfeiçoar a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos. Em essência, a refabricação pode ser usada para melhorar a eficiência, a legibilidade ou o desempenho de um projeto ou o código que implementa um projeto.

necessidades futuras podem mudar dramaticamente exigindo, conseqüentemente, substancial retrabalho em relação ao projeto e ao código implementado.

Ao seguir cada um desses valores, a equipe ágil inculca *respeito* entre seus membros, entre outros envolvidos e os membros da equipe, e, indiretamente, para o próprio software. Conforme conseguem entregar com sucesso incrementos de software, a equipe desenvolve cada vez mais respeito pelo processo XP.

3.4.2 O Processo XP

WebRef

Uma excelente visão geral das “regras” para XP pode ser encontrada em www.extremeprogramming.org/rules.html.

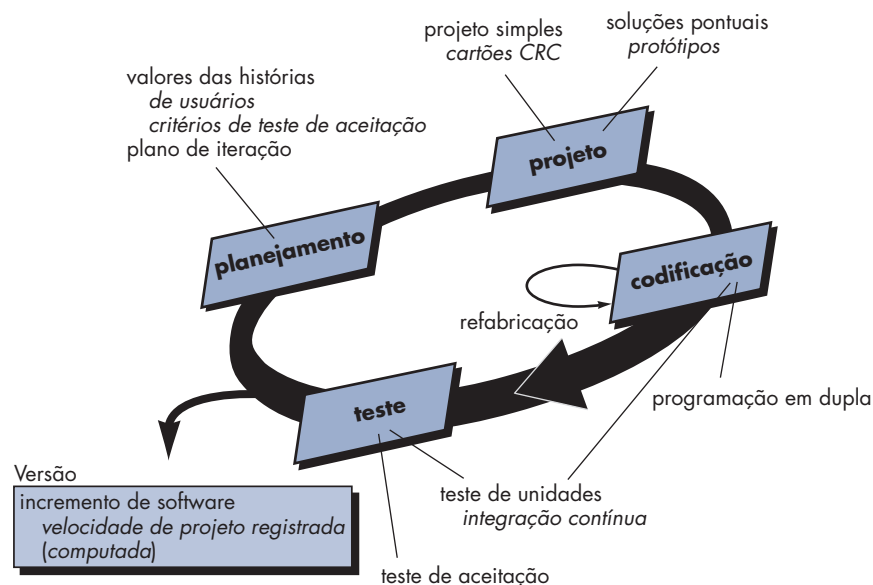
A *Extreme Programming* (programação extrema) emprega uma abordagem orientada a objetos (Apêndice 2) como seu paradigma de desenvolvimento preferido e envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes. A Figura 3.2 ilustra o processo XP e destaca alguns conceitos e tarefas-chave associados a cada uma das atividades metodológicas. As atividades-chave da XP são sintetizadas nos parágrafos a seguir.

Planejamento. A atividade de planejamento (também denominada *o jogo do planejamento*) se inicia com a atividade de *ouvir* — uma atividade de levantamento de requisitos que capacita os membros técnicos da equipe XP a entender o ambiente de negócios do software e possibilita que se consiga ter uma percepção ampla sobre os resultados solicitados, fatores principais e funcionalidade

A atividade de “Ouvir” conduz à criação de um conjunto de “histórias” (também denominado *histórias de usuários*) que descreve o resultado, as características e a funcionalidade requisitados para o software a ser construído. Cada *história* (similar aos casos de uso descritos no Capítulo 5) é escrita pelo cliente e é colocada em uma ficha. O cliente atribui um *valor* (uma prioridade) à história baseando-se no valor de negócio global do recurso ou função.⁵ Os membros da equipe XP avaliam então cada história e atribuem um *custo* — medido em semanas de desenvolvimento — a ela. Se a história requerer, por estimativa, mais do que três semanas de desenvolvimento, é solicitado ao cliente para dividir a história em histórias menores e a atribuição de valor e custo ocorre novamente. É importante notar que podem ser escritas novas histórias a qualquer momento.

? O que é uma “história” XP?

FIGURA 3.2
O processo da
Extreme
Programming (XP)



5 O valor de uma história também pode depender da presença de uma outra história.

WebRef

Um “jogo de planejamento” XP bastante interessante pode ser encontrado em: c2.com/cgi/wiki?planningGame.

PONTO-CHAVE

A velocidade do projeto é uma medida sutil da produtividade de uma equipe.



A XP tira a ênfase da importância do projeto. Nem todos concordam. De fato, há ocasiões em que o projeto deve ser enfatizado.

WebRef

Técnicas de refabricação e ferramentas podem ser encontradas em: www.refactoring.com.

PONTO-CHAVE

A refabricação aprimora a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos.

Clientes e desenvolvedores trabalham juntos para decidir como agrupar histórias para a versão seguinte (o próximo incremento de software) a ser desenvolvida pela equipe XP. Conseguindo chegar a um *compromisso básico* (concordância sobre quais histórias serão incluídas, data de entrega e outras questões de projeto) para uma versão, a equipe XP ordena as histórias a ser desenvolvidas em uma das três formas: (1) todas serão implementadas imediatamente (em um prazo de poucas semanas), (2) as histórias de maior valor serão deslocadas para cima no cronograma e implementadas primeiro ou (3) as histórias de maior risco serão deslocadas para cima no cronograma e implementadas primeiro.

Depois de a primeira versão do projeto (também denominada incremento de software) ter sido entregue, a equipe XP calcula a velocidade do projeto. De forma simples, a *velocidade do projeto* é o número de histórias de clientes implementadas durante a primeira versão. Assim, a velocidade do projeto pode ser utilizada para (1) ajudar a estimar as datas de entrega e o cronograma para versões subsequentes e (2) determinar se foi assumido um compromisso exagerado para todas as histórias ao longo de todo o projeto de desenvolvimento. Se ocorrer um exagero, o conteúdo das versões é modificado ou as datas finais de entrega são alteradas.

Conforme o trabalho de desenvolvimento prossegue, o cliente pode acrescentar histórias, mudar o valor de uma existente, dividir algumas ou eliminá-las. Em seguida, a equipe XP reconsidera todas as versões remanescentes e modifica seus planos de acordo.

Projeto. O projeto XP segue rigorosamente o princípio KIS (*keep it simple*, ou seja, preserve a simplicidade). É preferível sempre um projeto simples do que uma representação mais complexa. Como acréscimo, o projeto oferece um guia de implementação para uma história à medida que é escrita — nada mais, nada menos. O projeto de funcionalidade extra (pelo fato de o desenvolvedor supor que ela será necessária no futuro) é desencorajado.⁶

A XP encoraja o uso de cartões CRC (Capítulo 7) como um mecanismo eficaz para pensar sobre o software em um contexto orientado a objetos. Os cartões CRC (classe-responsabilidade-colaborador) identificam e organizam as classes orientadas a objetos⁷ relevantes para o incremento de software corrente. A equipe XP conduz o exercício de projeto usando um processo similar ao descrito no Capítulo 8. Os cartões CRC são o único artefato de projeto produzidos como parte do processo XP.

Se um difícil problema de projeto for encontrado como parte do projeto de uma história, a XP recomenda a criação imediata de um protótipo operacional dessa parte do projeto. Denominada *solução pontual*, o protótipo do projeto é implementado e avaliado. O objetivo é reduzir o risco para quando a verdadeira implementação iniciar e validar as estimativas originais para a história contendo o problema de projeto.

Na seção anterior, foi feita a observação de que a XP encoraja a *refatoração* — uma técnica de construção que também é um método para otimização de projetos. Fowler [Fow00] descreve a refabricação da seguinte maneira:

Refabricação é o processo de alteração de um sistema de software de tal forma que não se altere o comportamento externo do código, mas se aprimore a estrutura interna. É uma forma disciplinada de organizar código [e modificar/simplificar o projeto interno] que minimiza as chances de introdução de bugs. Em resumo, ao se refabricar, se está aperfeiçoando o projeto de codificação depois de este ter sido feito.

Como o projeto XP não usa praticamente nenhuma notação e produz poucos, se algum, artefatos, além dos cartões CRC e soluções pontuais, o projeto é visto como algo transitório que pode e deve ser continuamente modificado conforme a construção prossegue. O objetivo da refabricação é controlar tais modificações sugerindo pequenas mudanças de projeto “capazes de melhorá-lo radicalmente” [Fow00]. Deve ser observado, no entanto, que o esforço

⁶ Tais diretrizes de projeto deveriam ser seguidas em todos os métodos de engenharia de software, apesar de ocorrer situações em que sofisticadas terminologia e notação possam constituir obstáculo para a simplicidade.

⁷ As classes orientadas a objetos são discutidas no Apêndice 2, no Capítulo 8 e ao longo da Parte 2 deste livro.

necessário para a refabricação pode aumentar dramaticamente à medida que o tamanho de uma aplicação cresça.

Um aspecto central na XP é o de que a elaboração do projeto ocorre tanto antes *como depois* de se ter iniciado a codificação. Refabricação significa que o “projetar” é realizado continuamente enquanto o sistema estiver em elaboração. Na realidade, a própria atividade de desenvolvimento guiará a equipe XP quanto à aprimoração do projeto.

Codificação. Depois de desenvolvidas as histórias e o trabalho preliminar de elaboração do projeto ter sido feito, a equipe *não passa* para a codificação, mas sim, desenvolve uma série de testes de unidades que exercitarão cada uma das histórias a ser incluídas na versão corrente (incremento de software).⁸ Uma vez criado o teste de unidades⁹, o desenvolvedor poderá melhor focar-se no que deve ser implementado para ser aprovado no teste. Nada estranho é adicionado (KIS). Estando o código completo, este pode ser testado em unidade imediatamente, e, dessa forma, prover, instantaneamente, feedback para os desenvolvedores.

Um conceito-chave na atividade de codificação (e um dos mais discutidos aspectos da XP) é a *programação em dupla*. A XP recomenda que duas pessoas trabalhem juntas em uma mesma estação de trabalho para criar código para uma história. Isso fornece um mecanismo para resolução de problemas em tempo real (duas cabeças normalmente funcionam melhor do que uma) e garantia da qualidade em tempo real (o código é revisto à medida que é criado). Ele também mantém os desenvolvedores focados no problema em questão. Na prática, cada pessoa assume um papel ligeiramente diferente. Por exemplo, uma pessoa poderia pensar nos detalhes de codificação de determinada parte do projeto, enquanto outra assegura que padrões de codificação (uma parte exigida pela XP) sejam seguidos ou que o código para a história passará no teste de unidades desenvolvido para validação do código em relação à história.

Conforme a dupla de programadores completa o trabalho, o código que desenvolveram é integrado ao trabalho de outros. Em alguns casos, isso é realizado diariamente por uma equipe de integração. Em outros, a dupla de programadores é responsável pela integração. A estratégia de “integração contínua” ajuda a evitar problemas de compatibilidade e de interfaceamento, além de criar um ambiente “teste da fumaça” (Capítulo 17) que ajuda a revelar erros precocemente.

Testes. Já foi observado que a criação de testes de unidade, antes de começar a codificação, é um elemento-chave da abordagem XP. Os testes de unidade criados devem ser implementados usando-se uma metodologia que os capacite a ser automatizados (assim, poderão ser executados fácil e repetidamente). Isso encoraja uma estratégia de testes de regressão (Capítulo 17), toda vez em que o código for modificado (o que é frequente, dada a filosofia de refabricação da XP).

Como os testes de unidades individuais são organizados em um “conjunto de testes universal” [Wel99], os testes de integração e validação do sistema podem ocorrer diariamente. Isso dá à equipe XP uma indicação contínua do progresso e também permite lançar alertas logo no início, caso as coisas não andem bem. Wells [Wel99] afirma: “Corrigir pequenos problemas em intervalos de poucas horas leva menos tempo do que corrigir problemas enormes próximo ao prazo de entrega”.

Os *testes de aceitação* da XP, também denominados *testes de cliente*, são especificados pelo cliente e mantêm o foco nas características e na funcionalidade do sistema total que são visíveis e que podem ser revistas pelo cliente. Os testes de aceitação são obtidos de histórias de usuários implementadas como parte de uma versão de software.

WebRef

Informações úteis sobre a XP podem ser obtidas em www.xprogramming.com.

? O que é programação em dupla?



Muitas equipes de software são constituídas por individualistas. Deverá haver empenho para modificar tal cultura, para que a programação em dupla funcione efetivamente.

? Como são usados os testes de unidade na XP?

PONTO-CHAVE

Os testes de aceitação da XP são elaborados com base nas histórias de usuários.

⁸ Essa abordagem é como conhecer as perguntas de uma prova antes de começar a estudar. Torna o estudo muito mais fácil, permitindo que se concentre a atenção apenas nas perguntas que serão feitas.

⁹ O teste de unidades, discutido detalhadamente no Capítulo 17, concentra-se em um componente de software individual, exercitando a interface, a estrutura de dados e a funcionalidade do componente, em uma tentativa de que se revelem erros pertinentes ao componente.

3.4.3 Industrial XP

Joshua Kerievsky [Ker05] descreve a *Industrial Extreme Programming* (IXP) — programação extrema industrial — da seguinte maneira: “A IXP é uma evolução orgânica da XP. Ela é imbuída do espírito minimalista, centrado no cliente e orientado a testes da XP. Difere principalmente da XP original por sua maior inclusão do gerenciamento, por seu papel expandido para os clientes e por suas práticas técnicas atualizadas”. A IXP incorpora seis novas práticas desenvolvidas para ajudar a assegurar que um projeto XP funcione com êxito em empreendimentos significativos em uma grande organização.

 **Que novas práticas são acrescentadas à XP para elaborar a IXP?**

“Habilidade consiste no que se é capaz de fazer. Motivação determina o que você faz. Atitude determina quão bem você faz.”

Lou Holtz

Avaliação imediata. Antes do início de um projeto IXP, a organização deve realizar uma *avaliação imediata*. A avaliação verifica se (1) existe um ambiente de desenvolvimento apropriado para sustentar a IXP, (2) a equipe será composta por um conjunto apropriado de interessados, (3) a organização possui um programa de qualidade diferenciado e suporta contínuo aperfeiçoamento, (4) a cultura organizacional apoiará os novos valores de uma equipe ágil e (5) a comunidade de projeto ampliada será composta apropriadamente.

Comunidade de projeto. A XP clássica sugere que se aloquem as pessoas acertadas para compor a equipe ágil e garantir o sucesso. Isso implica pessoas da equipe bem treinadas, adaptáveis e experientes e que tenham temperamento apropriado para contribuir para uma equipe auto-organizada. Ao se aplicar a XP em um projeto importante de uma grande empresa, o conceito da “equipe” deve transformar-se no de *comunidade*. A comunidade pode ter um tecnólogo e clientes fundamentais para o sucesso de um projeto, assim como muitos outros envolvidos (por exemplo, responsáveis jurídicos, auditores do controle da qualidade, representantes da área de produção ou de categorias de vendas) que “frequentemente se encontram na periferia de um projeto IXP, mas que podem desempenhar importante papel no projeto” [Ker05]. Na IXP, os membros da comunidade devem ter papéis explicitamente definidos e os mecanismos de comunicação e de coordenação relativos aos elementos da comunidade devem estar determinados.

Mapeamento do projeto. A própria equipe IXP avalia o projeto para determinar se este se justifica em termos de negócios e se irá ultrapassar as metas e objetivos globais da organização. O mapeamento também examina o contexto do projeto para estabelecer como este complementa, amplia ou substitui sistemas ou processos existentes.

Gerenciamento orientado a testes. Um projeto IXP requer critérios mensuráveis para avaliar o estado do projeto e do progresso obtido até então. O gerenciamento orientado a testes estabelece uma série de “destinos” mensuráveis [Ker05] e define mecanismos para determinar se estes foram atingidos ou não.

Retrospectivas. Uma equipe IXP conduz uma revisão técnica especializada (Capítulo 15) após a entrega de um incremento de software. Denominada *retrospectiva*, a revisão examina “itens, eventos e lições aprendidas” [Ker05] ao longo do processo de incremento de software e/ou do desenvolvimento da versão completa do software. O objetivo é aprimorar o processo da IXP.

Aprendizagem contínua. Sendo a aprendizagem uma parte vital para o aperfeiçoamento contínuo do processo, os membros da equipe XP são encorajados (e possivelmente incentivados) a aprender novos métodos e técnicas que possam conduzir a um produto de melhor qualidade.

Somando-se às apresentadas, a IXP modifica uma série de práticas XP existentes. O desenvolvimento orientado por histórias (*story-driven development*, *SDD*) insiste que as histórias para testes de aceitação sejam escritas antes de gerar uma única linha de código. O projeto orientado por domínio (*domain-driven design*, *DDD*) é um aprimoramento do conceito “metáfora de sistema” usado na XP. O DDD [Eva03] sugere a criação evolucionária de um modelo de domínio que “represente acuradamente como pensam os especialistas de determinado domínio dentro de sua disciplina” [Ker05]. O *emparelhamento* amplia o conceito de programação em dupla da

XP, ao incluir gerentes e outros envolvidos. O intuito é ampliar o compartilhamento de conhecimentos entre os membros da equipe XP que possam não estar diretamente envolvidos no desenvolvimento técnico. A *usabilidade iterativa* desencoraja o projeto de interfaces de carregamento frontal (*front-loaded interface design*), sendo a favor do projeto de usabilidade que evolui conforme os incrementos sejam entregues e a interação entre usuários e o software seja estudada.

A IXP faz modificações menores para outras práticas XP e redefine certos papéis e responsabilidades para torná-los mais harmonizados com projetos importantes de organizações. Para uma discussão mais ampla sobre a IXP, visite <http://industrialxp.org>.

3.4.4 O Debate XP

Todos os novos métodos e modelos de processos estimulam debates úteis e, em alguns casos, debates acalorados. A *Extreme Programming* provocou ambos. Em um livro interessante que examina a eficácia da XP, Stephens e Rosenberg [Ste03] argumentam que muitas práticas XP valem a pena, mas outras foram superestimadas e algumas poucas são problemáticas. Os autores sugerem que a codependência da prática da XP representa sua força e sua fraqueza. Pelo fato de muitas organizações adotarem apenas um subconjunto de práticas XP, elas enfraquecem a eficácia de todo o processo. Seus defensores rebatem dizendo que a XP é aperfeiçoada continuamente e que muitos dos itens levantados pela crítica têm sido acessados conforme a prática da XP ganha maturidade. Entre os itens que continuam a incomodar certos críticos da XP estão:¹⁰

- *Volatilidade de requisitos.* Pelo fato de o cliente ser um membro ativo da equipe XP, alterações de requisitos são solicitadas informalmente. Como consequência, o escopo do projeto pode mudar e trabalhos anteriores podem ter de vir a ser alterados, a fim de acomodar as necessidades de então. Seus defensores argumentam que isso acontece independentemente do processo aplicado e que a XP oferece mecanismos para controlar o surgimento incontrolado de novos escopos.
- *Necessidades conflitantes de clientes.* Projetos em quantidade possuem múltiplos clientes, cada um com seu próprio conjunto de necessidades. Na XP, a própria equipe tem a tarefa de assimilar as necessidades de diferentes clientes, um trabalho que pode estar além de seu escopo de autoridade.
- *Os requisitos são levantados informalmente.* Histórias de usuários e testes de aceitação são a única manifestação explícita de requisitos da XP. Seus críticos argumentam que, frequentemente, torna-se necessário um modelo ou especificação mais formal para assegurar que omissões, inconsistências e erros sejam descobertos antes que o sistema seja construído. Seus defensores rebatem dizendo que a natureza mutante de requisitos torna tais modelos e especificações obsoletos praticamente logo depois de terem sido desenvolvidos.
- *Falta de projeto formal.* A XP tira a ênfase da necessidade do projeto de arquitetura e, em muitos casos, sugere que todos os tipos de projetos devam ser relativamente informais. Seus críticos argumentam que em sistemas complexos deve-se enfatizar a elaboração do projeto para assegurar que a estrutura geral do software apresentará qualidade e facilidade de manutenção. Já os defensores da XP sugerem que a natureza incremental do processo XP limita a complexidade (a simplicidade é um valor fundamental) e, consequentemente, reduz a necessidade de um projeto extenso.

Deve-se observar que todo processo de software tem suas falhas e que muitas organizações de software usaram, com êxito, a XP. O segredo é reconhecer onde um processo pode apresentar fraquezas e adaptá-lo às necessidades específicas de sua organização.

 Quais são alguns dos pontos que conduzem a um debate a respeito da XP?

10 Para uma visão detalhada de algumas críticas ponderadas feitas ao XP, visite www.softwarereality.com/ExtremeProgramming.jsp.

CASA SEGURA



Considerando o desenvolvimento de software ágil

Cena: de Doug Miller.

Atores: Doug Miller, gerente de engenharia de software; Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software.

Conversa:

(Batendo à porta, Jamie e Vinod adentram à sala de Doug)

Jamie: Doug, você tem um minuto?

Doug: Com certeza, Jamie, o que há?

Jamie: Estivemos pensando a respeito da discussão sobre processos, de ontem... Sabe, que processo vamos escolher para este novo projeto *CasaSegura*.

Doug: E?

Vinod: Eu estava conversando com um amigo de uma outra empresa e ele me falou sobre a Extreme Programming. É um modelo de processo ágil... Já ouviu falar?

Doug: Sim, algumas coisas boas, outras ruins.

Jamie: Bem, pareceu muito bom para nós. Permite que se desenvolva software realmente rápido, usa algo chamado programação em dupla para fazer checagens de qualidade em tempo real... É bem legal, eu acho.

Doug: Realmente, apresenta um monte de ideias muito boas. Gosto do conceito de programação em dupla, por exemplo, e da ideia de que os envolvidos devam fazer parte da equipe.

Jamie: O quê? Quer dizer que o pessoal de marketing trabalhará conosco na equipe de projeto?

Doug (confirmando com a cabeça): Eles são envolvidos, não são?

Jamie: Jesus... Eles solicitarão alterações a cada cinco minutos.

Vinod: Não necessariamente. Meu amigo me disse que existem formas de se “abarcas” as mudanças durante um projeto XP.

Doug: Portanto, meus amigos, vocês acham que deveríamos usar a XP?

Jamie: Definitivamente, vale considerar.

Doug: Eu concordo. E mesmo que optássemos por um modelo incremental como nossa abordagem, não há nenhuma razão para não podermos incorporar muito do que a XP tem a oferecer.

Vinod: Doug, mas antes você disse “algumas coisas boas, outras ruins”. O que são as “coisas ruins”?

Doug: O que não me agrada é a maneira pela qual a XP dá menos importância à análise e ao projeto... Dizem algo como: a codificação resume a ação para construir um software.

(Os membros da equipe se entreolham e sorriem.)

Doug: Então vocês concordam com a abordagem XP?

Jamie (falando por ambos): Escrever código é o que fazemos, chefe!

Doug (rindo): É verdade, mas eu gostaria de vê-lo perdendo um pouco menos de tempo codificando para depois recodificar e dedicando um pouco mais de tempo analisando o que precisa ser feito e projetando uma solução que funcione.

Vinod: Talvez possamos ter as duas coisas, agilidade com um pouco de disciplina.

Doug: Acho que sim, Vinod. Na realidade, tenho certeza disso.

3.5 OUTROS MODELOS DE PROCESSOS ÁGEIS

“Nossa profissão passa por metodologias como uma garota de 14 anos passa por roupas.”

Stephen Hawrysh e Jim Ruprecht

Na história da engenharia de software há dezenas de metodologias e descrições de processos, métodos e notações de modelagem, ferramentas e tecnologias obsoletas. Cada um atingiu grande notoriedade e foi então ofuscado por algo novo e (supostamente) melhor. Com a introdução de uma ampla gama de modelos de processos ágeis — todos disputando por aceitação pela comunidade de desenvolvimento de software —, o movimento ágil está seguindo o mesmo caminho histórico.¹¹

Conforme citado na última seção, o modelo mais amplamente utilizado de todos os modelos de processos ágeis é o da Extreme Programming (XP). Porém, muitos outros têm sido propostos e encontram-se em uso no setor. Entre os mais comuns, temos:

- Desenvolvimento de software adaptativo (Adaptive Software Development, ASD)
- Scrum
- Método de desenvolvimento de sistemas dinâmicos (Dynamic Systems Development Method, DSDM)
- Crystal

¹¹ Isso não é algo ruim. Antes que um ou mais modelos ou métodos sejam aceitos como um padrão *de fato*, todos devem competir para conquistar os corações e mentes dos engenheiros de software. Os “vencedores” evoluem e se transformam nas melhores práticas, enquanto os “perdedores” desaparecem ou se fundem aos modelos vencedores.

- Desenvolvimento dirigido a Funcionalidades (Feature Drive Development, FDD)
- Desenvolvimento de software enxuto (Lean Software Development, LSD)
- Modelagem ágil (Agile Modeling, AM)
- Processo unificado ágil (Agile Unified Process, AUP)

Nas seções seguintes, apresenta-se uma visão geral muito breve de cada um desses modelos de processos ágeis. É importante observar que *todos* estão em conformidade (em maior ou menor grau) com o *Manifesto for Agile Software Development* e com os princípios citados na Seção 3.3.1. Para mais detalhes, veja as referências citadas em cada subseção ou, para uma pesquisa, examine a entrada “agile software development” na Wikipedia.¹²

3.5.1 Desenvolvimento de Software Adaptativo (ASD)

O desenvolvimento de software adaptativo (*Adaptive Software Development*) foi proposto por Jim Highsmith [Hig00] como uma técnica para construção de software e sistemas complexos. As bases filosóficas do ASD se concentram na colaboração humana e na auto-organização das equipes.

Highsmith argumenta que uma abordagem de desenvolvimento ágil e adaptativo baseada na colaboração constitui “um recurso para organizar nossas complexas interações, tanto quanto disciplina e engenharia o são”. Ele define um “ciclo de vida” ASD (Figura 3.3) que incorpora três fases: especulação, colaboração e aprendizagem.

Durante a *especulação*, o projeto é iniciado e conduzido o *planejamento de ciclos adaptáveis*. O *planejamento de ciclos adaptáveis* usa as informações do início de projeto — o estabelecimento da missão do cliente, as restrições do projeto (por exemplo, datas de entrega ou descrições de usuários) e os requisitos básicos — para definir o conjunto de ciclos de versão (incrementos de software) que serão requisitados para o projeto.

Não importa quão completo e com visão de futuro seja o plano de ciclos, invariavelmente sofrerá mudanças. Baseando-se nas informações obtidas ao se completar o primeiro ciclo, o plano é revisto e ajustado de modo que o trabalho planejado melhor se ajuste à realidade na qual a equipe ASD está trabalhando.

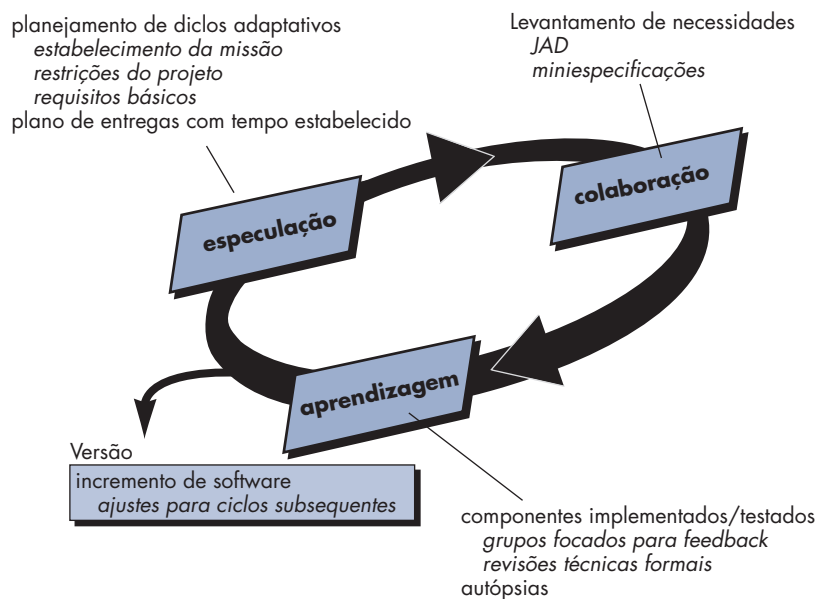
WebRef

Recursos úteis para ASD podem ser encontrados em www.adaptivesd.com.



A colaboração efetiva com seu cliente ocorrerá somente se você extinguir quaisquer atitudes de “nós e eles”.

FIGURA 3.3
Desenvolvimento de software adaptável



¹² Veja http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

As pessoas motivadas usam a *colaboração* de uma forma que multiplique seus talentos e produções criativas além de seus números absolutos. Tal abordagem é tema recorrente em todos os métodos ágeis. Porém, colaboração não é algo fácil, envolve comunicação e trabalho em equipe, mas também enfatiza o individualismo, pois a criatividade individual desempenha um importante papel no pensamento colaborativo. Trata-se, sobretudo, de uma questão de confiança. Pessoas que trabalham juntas têm de confiar umas nas outras para (1) criticar sem animosidade, (2) auxiliar sem ressentimentos, (3) trabalhar tão arduamente ou mais do que elas fazem, (4) possuir o conjunto de habilidades que contribua com o atual trabalho e (5) comunicar problemas ou preocupações de forma que conduzam a ações efetivas.

PONTO-CHAVE

O ASD enfatiza o aprendizado como elemento-chave para conseguir uma equipe “auto-organizada”.

Conforme os membros de uma equipe ASD comecem a desenvolver os componentes que fazem parte de um ciclo adaptável, a ênfase reside no “aprendizado” tanto quanto reside no progresso para um ciclo completado. De fato, Highsmith [Hig00] argumenta que os desenvolvedores de software normalmente superestimam seu próprio entendimento (da tecnologia, do processo e do projeto) e que a aprendizagem irá ajudá-los a aumentar seus níveis reais de entendimento. As equipes ASD aprendem de três maneiras: grupos focados (Capítulo 5), revisões técnicas (Capítulo 14) e autópsias de projetos (análises *postmortems*).

A filosofia ASD tem seus méritos independentemente do modelo de processos utilizado. A ênfase global da ASD está na dinâmica das equipes auto-organizadas, na colaboração interpessoal e na aprendizagem individual e da equipe que levam as equipes de projeto de software a uma probabilidade muito maior de sucesso.

3.5.2 Scrum

WebRef

Informações e recursos úteis sobre o Scrum podem ser encontrados em www.controlchaos.com.

Scrum (o nome provém de uma atividade que ocorre durante a partida de rugby¹³) é um método de desenvolvimento ágil de software concebido por Jeff Sutherland e sua equipe de desenvolvimento no início dos anos 1990. Mais recentemente, foram realizados desenvolvimentos adicionais nos métodos gráficos Scrum por Schwaber e Beedle [Sch01a].

Os princípios do Scrum são consistentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades estruturais: requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica, ocorrem tarefas a realizar dentro de um padrão de processo (discutido no parágrafo a seguir) chamado *sprint*. O trabalho realizado dentro de um sprint (o número de sprints necessários para cada atividade metodológica varia dependendo do tamanho e da complexidade do produto) é adaptado ao problema em questão e definido, e muitas vezes modificado em tempo real, pela equipe Scrum. O fluxo geral do processo Scrum é ilustrado na Figura 3.4.

O Scrum enfatiza o uso de um conjunto de padrões de processos de software [Noy02] que provaram ser eficazes para projetos com prazos de entrega apertados, requisitos mutáveis e críticos de negócio. Cada um desses padrões de processos define um conjunto de ações de desenvolvimento:

Registro pendente de trabalhos (Backlog) — uma lista com prioridades dos requisitos ou funcionalidades do projeto que fornecem valor comercial ao cliente. Os itens podem ser adicionados a esse registro em qualquer momento (é assim que as alterações são introduzidas). O gerente de produto avalia o registro e atualiza as prioridades conforme requisitado.

Urgências (corridas de curta distância) sprints — consistem de unidades de trabalho solicitadas para atingir um requisito estabelecido no registro de trabalho (backlog) e que precisa ser ajustado dentro de um prazo já fechado (janela de tempo)¹⁴ (tipicamente 30 dias).

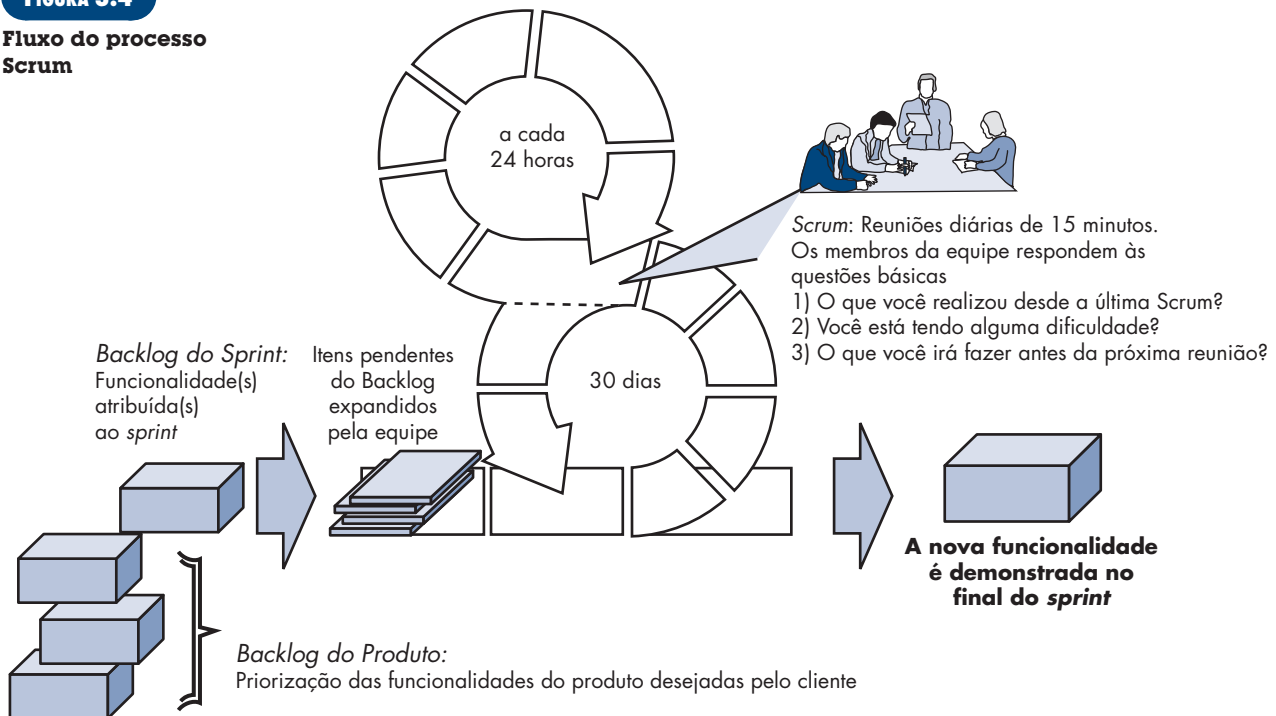
Alterações (por exemplo, itens do registro de trabalho — *backlog work itens*) não são introduzidas durante execução de urgências (*sprint*). Portanto, o *sprint* permite que os membros de uma equipe trabalhem em um ambiente de curto prazo, porém estável.

PONTO-CHAVE

O Scrum engloba um conjunto de padrões de processos enfatizando prioridades de projeto, unidades de trabalho compartimentalizadas, comunicação e feedback frequente por parte dos clientes.

13 Um grupo de jogadores faz uma formação em torno da bola e seus companheiros de equipe trabalham juntos (às vezes, de forma violenta!) para avançar com a bola em direção ao fundo do campo.

14 *Janela de tempo (time boxing)* é um termo de gerenciamento de projetos (veja a Parte 4 deste livro) que indica um período de tempo destinado para cumprir alguma tarefa.

FIGURA 3.4**Fluxo do processo Scrum**

Reuniões Scrum — são reuniões curtas (tipicamente 15 minutos), realizadas diariamente pela equipe Scrum. São feitas três perguntas-chave e respondidas por todos os membros da equipe [Noy02]:

- O que você realizou desde a última reunião de equipe?
- Quais obstáculos está encontrando?
- O que planeja realizar até a próxima reunião da equipe?

Um líder de equipe, chamado *Scrum master*, conduz a reunião e avalia as respostas de cada integrante. A reunião Scrum, realizada diariamente, ajuda a equipe a revelar problemas potenciais o mais cedo possível. Ela também leva à “socialização do conhecimento” [Bee99] e, portanto, promove uma estrutura de equipe auto-organizada.

Demos — entrega do incremento de software ao cliente para que a funcionalidade implementada possa ser demonstrada e avaliada pelo cliente. É importante notar que a demo pode não ter toda a funcionalidade planejada, mas sim funções que possam ser entregues no prazo estipulado.

Beedle e seus colegas [Bee99] apresentam uma ampla discussão sobre esses padrões: “O Scrum pressupõe a existência do caos...”. Os padrões de processos do Scrum capacitam uma equipe de software a trabalhar com sucesso em um mundo onde a eliminação da incerteza é impossível.

3.5.3 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM)

O *método de desenvolvimento de sistemas dinâmicos* (Dynamic Systems Development Method) [Sta97] é uma abordagem de desenvolvimento de software ágil que “oferece uma metodologia para construir e manter sistemas que atendem restrições de prazo apertado através do uso da prototipagem incremental em um ambiente de projeto controlado” [CCS02]. A filosofia DSDM baseia-se em uma versão modificada do princípio de Pareto — 80% de uma aplicação pode ser entregue em 20% do tempo que levaria para entregar a aplicação completa (100%).

WebRef

Recursos úteis para o DSSD podem ser encontrados em www.dsdm.org.

O DSDM é um processo de software iterativo em que cada iteração segue a regra dos 80%. Ou seja, somente o trabalho suficiente é requisitado para cada incremento, para facilitar o movimento para o próximo incremento. Os detalhes remanescentes podem ser completados depois, quando outros requisitos de negócio forem conhecidos ou alterações tiverem sido solicitadas e acomodadas.

O DSDM Consortium (www.dsdm.org) é um grupo mundial de empresas-membro que coletivamente assume o papel de “mantenedor” do método. Esse consórcio definiu um modelo de processos ágeis, chamado *ciclo de vida DSDM* que define três ciclos iterativos diferentes, precedidos por duas atividades de ciclo de vida adicionais:

Estudo da viabilidade — estabelece os requisitos básicos de negócio e restrições associados à aplicação a ser construída e depois avalia se a aplicação é ou não um candidato viável para o processo DSDM.

Estudo do negócio — estabelece os requisitos funcionais e de informação que permitirão à aplicação agregar valor de negócio; define também a arquitetura básica da aplicação e identifica os requisitos de facilidade de manutenção para a aplicação.

Iteração de modelos funcionais — produz um conjunto de protótipos incrementais que demonstram funcionalidade para o cliente. (Nota: Todos os protótipos DSDM são feitos com a intenção de que evoluam para a aplicação final entregue ao cliente.) Durante esse ciclo iterativo, o objetivo é juntar requisitos adicionais ao se obter feedback dos usuários, conforme testam o protótipo.

Iteração de projeto e desenvolvimento — revisita protótipos desenvolvidos durante a *iteração de modelos funcionais* para assegurar-se de que cada um tenha passado por um processo de engenharia para capacitá-los a oferecer, aos usuários finais, valor de negócio em termos operacionais. Em alguns casos, a *iteração de modelos funcionais* e a *iteração de projeto e desenvolvimento* ocorrem ao mesmo tempo.

Implementação — aloca a última versão do incremento de software (um protótipo “operacionalizado”) no ambiente operacional. Deve-se notar que: (1) o incremento pode não estar 100% completo ou (2) alterações podem vir a ser solicitadas conforme o incremento seja alocado. Em qualquer dos casos, o trabalho de desenvolvimento do DSDM continua, retornando-se à atividade de iteração do modelo funcional.

O DSDM pode ser combinado com a XP (Seção 3.4) para fornecer uma abordagem combinatória que define um modelo de processos consistente (o ciclo de vida do DSDM) com as práticas básicas (XP) necessárias para construir incrementos de software. Além disso, os conceitos de colaboração e de equipes auto-organizadas do ASD podem ser adaptados a um modelo de processos combinado.

3.5.4 Crystal

Alistair Cockburn [Coc05] e Jim Highsmith [Hig02b] criaram a *família Crystal de métodos ágeis*¹⁵ visando conseguir elaborar uma abordagem de desenvolvimento de software que priorizasse a adaptabilidade (“maneuverability”) durante o que Cockburn caracteriza como um “jogo de invenção e comunicação cooperativo e com recursos limitados, tendo como primeiro objetivo entregar software útil em funcionamento e como segundo objetivo preparar-se para o jogo seguinte” [Coc02].

Para conseguir adaptabilidade, Cockburn e Highsmith definiram um conjunto de metodologias com elementos essenciais comuns a todas, mas com papéis, padrões de processos, produtos de trabalho e prática únicos para cada uma delas. A família Crystal é, na verdade, um conjunto de exemplos de processos ágeis que provaram ser efetivos para diferentes tipos de projetos. A intenção é possibilitar que equipes ágeis selecionem o membro da família Crystal mais apropriado para seu projeto e seu ambiente.

PONTO-CHAVE

Crystal é uma família de modelos de processos com o mesmo “código genético”, mas com diferentes métodos para se adaptarem às características do projeto.

PONTO-CHAVE

O DSDM é um metodologia de processos que pode adotar a tática de uma outra abordagem ágil como a XP.

¹⁵ O nome “crystal” (cristal) é derivado das características dos cristais geológicos, cada qual com sua cor, forma e dureza próprias.

3.5.5 Desenvolvimento Dirigido a Funcionalidades (FDD)

O desenvolvimento dirigido a funcionalidades (*Feature Driven Development*) foi concebido originalmente por Peter Coad e seus colegas [Coa99] como um modelo de processos prático para a engenharia de software orientada a objetos. Stephen Palmer e John Felsing [Pal02] estenderam e aperfeiçoaram o trabalho de Coad, descrevendo um processo ágil adaptativo que pode ser aplicado a projetos de software de porte moderado e a projetos maiores.

WebRef

Uma ampla variedade de artigos e apresentações sobre o FDD pode ser encontrada em: www.featuredrivendevelopment.com/.

Como outras abordagens ágeis, o FDD adota uma filosofia que (1) enfatiza a colaboração entre pessoas da equipe FDD; (2) gerencia problemas e complexidade de projetos utilizando a decomposição baseada em funcionalidades, seguida pela integração dos incrementos de software, e (3) comunicação de detalhes técnicos usando meios verbais, gráficos e de texto. O FDD enfatiza as atividades de garantia da qualidade de software por meio do encorajamento de uma estratégia de desenvolvimento incremental, o uso inspeções do código e do projeto, a aplicação de auditorias para garantia da qualidade de software (Capítulo 16), a coleta de métricas e o uso de padrões (para análise, projeto e construção).

No contexto do FDD, funcionalidade “é uma função valorizada pelo cliente passível de ser implementada em duas semanas ou menos” [Coa99]. A ênfase na definição de funcionalidades gera os seguintes benefícios:

- Como as funcionalidades formam pequenos blocos que podem ser entregues, os usuários podem descrevê-las mais facilmente; compreender como se relacionam entre si mais prontamente; e revisá-las melhor em termos de ambiguidade, erros ou omissões.
- As funcionalidades podem ser organizadas em um agrupamento hierárquico relacionado com o negócio.
- Como uma funcionalidade é o incremento de software do FDD que pode ser entregue, a equipe desenvolve funcionalidades operacionais a cada duas semanas.
- Pelo fato de o bloco de funcionalidades ser pequeno, seus projeto e representações de código são mais fáceis de inspecionar efetivamente.
- O planejamento, cronograma e acompanhamento do projeto são guiados pela hierarquia de funcionalidades, em vez de um conjunto de tarefas de engenharia de software arbitrariamente adotado.

Coad e seus colegas [Coa99] sugerem o seguinte modelo para definir uma funcionalidade:

<acao> o <resultado> <por| para quem |de |para que> um <objeto>

em que um **<objeto>** é “uma pessoa, local ou coisa (inclusive papéis, momentos no tempo ou intervalos de tempo ou descrições parecidas com aquelas encontradas em catálogos)”. Exemplos de funcionalidades para uma aplicação de comércio eletrônico poderiam ser:

Adicione o produto ao carrinho
Mostre as especificações técnicas do produto
Armazene as informações de remessa para o cliente

Um conjunto de funcionalidades agrupa funcionalidades em categorias correlacionadas por negócio e é definido [Coa99] com:

<acao> um <objeto>

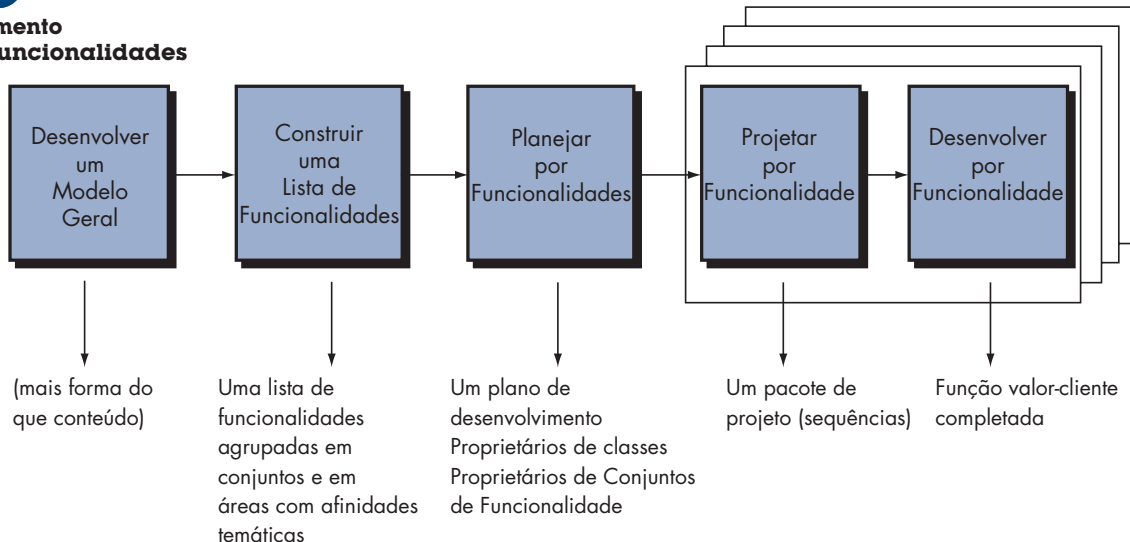
Por exemplo: *Fazer a venda de um produto* é um conjunto de funcionalidades que abrangeria os fatores percebidos anteriormente e outros.

A abordagem FDD define cinco atividades metodológicas “colaborativas” [Coa99] (no FDD estas são denominadas “processos”) conforme mostra a Figura 3.5.

O FDD oferece maior ênfase às diretrizes e técnicas de gerenciamento de projeto do que muitos outros métodos ágeis. Conforme os projetos crescem em tamanho e complexidade, com

FIGURA 3.5

**Desenvolvimento
dirigido a funcionalidades
(Coa99)
(com
permissão)**



frequência o gerenciamento de projeto para finalidade local torna-se inadequado. É essencial para os desenvolvedores, seus gerentes e outros envolvidos compreenderem o posicionamento do projeto — que realizações foram feitas e que problemas foram encontrados. Se a pressão do prazo de entrega for significativa, é crítico determinar se os incrementos de software (funcionalidades) foram agendados apropriadamente. Para tanto, o FDD define seis marcos durante o projeto e a implementação de uma funcionalidade: “desenrolar (*walkthroughs*) do projeto, projeto, inspeção de projeto, codificação, inspeção de código, progressão para construção/desenvolvimento” [Coa99].

3.5.6 Desenvolvimento de Software Enxuto (LSD)

O desenvolvimento de software enxuto (*Lean Software Development*) adaptou os princípios da fabricação enxuta para o mundo da engenharia de software. Os princípios enxutos que inspiraram o processo LSD podem ser sintetizados ([Pop03], [Pop06a]) em: *eliminar desperdício, incorporar qualidade, criar conhecimento, adiar compromissos, entregar rápido, respeitar as pessoas e otimizar o todo*.

Cada um dos princípios pode ser adaptado ao processo de software. Por exemplo, *eliminar desperdício* no contexto de um projeto de software ágil pode ser interpretado como [Das05]: (1) não adicionar recursos ou funções estranhas, (2) avaliar o impacto do custo e do cronograma de qualquer requisito solicitado recentemente, (3) eliminar quaisquer etapas de processo supérfluas, (4) estabelecer mecanismos para aprimorar o modo pelo qual a equipe levanta informações, (5) assegurar-se de que os testes encontrem o maior número de erros possível, (6) reduzir o tempo necessário para solicitar e obter uma decisão que afete o software ou o processo aplicado para criá-lo e (7) racionalizar a maneira pela qual informações são transmitidas a todos envolvidos no processo.

Para uma discussão detalhada do LSD e diretrizes pragmáticas para implementação do processo, consulte [Pop06a] e [Pop06b].

3.5.7 Modelagem Ágil (AM)

Existem muitas situações em que engenheiros de software têm de desenvolver sistemas grandes, com detalhes críticos em termos de negócio. O escopo e complexidade de tais sistemas devem ser modelados de modo que (1) todas as partes envolvidas possam entender melhor

WebRef

Informação ampla sobre a modelagem ágil pode ser encontrada em: www.agilemodeling.com.

“Um dia, estava em uma farmácia tentando achar um remédio para resfriado... Não foi fácil... Havia uma parede inteira de produtos. Fica-se lá procurando: ‘Bem, este tem ação imediata, mas este outro tem efeito mais duradouro...’. O que é mais importante, o presente ou o futuro?”

Jerry Seinfeld

quais requisitos deverão ser atingidos, (2) o problema possa ser subdividido eficientemente entre as pessoas que têm de solucioná-lo e (3) a qualidade possa ser avaliada enquanto se está projetando e desenvolvendo o sistema.

Ao longo dos últimos 30 anos, uma ampla variedade de notações e métodos de modelagem de engenharia de software tem sido proposta para análise e projeto (tanto no nível de componente como de arquitetura). Esses métodos têm seus méritos, mas provaram ser difíceis de ser aplicados e desafiadores para ser mantidos (ao longo de vários projetos). Parte do problema é o “peso” dos métodos de modelagem. Com isso quero dizer o volume de notação exigido, o grau de formalismo sugerido, o puro tamanho dos modelos para grandes projetos e a dificuldade em manter o(s) modelo(s) à medida que ocorrem as mudanças. Contudo, o modelamento de análise e projeto tem um benefício substancial para grandes projetos — ainda que seja apenas para torná-los intelectualmente gerenciáveis. Existe uma abordagem ágil para a modelagem de engenharia de software que poderia fornecer uma alternativa?

No “The Official Agile Modeling Site”, Scott Ambler [Amb02a] descreve *modelagem ágil (AM)* da seguinte maneira:

Modelagem ágil (AM) consiste em uma metodologia baseada na prática, voltada para o modelamento e documentação de sistemas com base em software. Simplificando, modelagem ágil consiste em um conjunto de valores, princípios e práticas voltados para a modelagem do software que pode ser aplicados em um projeto de desenvolvimento de software de forma leve e efetiva. Os modelos ágeis são mais efetivos do que os tradicionais pelo fato de serem meramente bons, pois não têm a obrigação de ser perfeitos.

Modelagem ágil adota todos os valores consistentes com o manifesto ágil. Sua filosofia reconhece que uma equipe ágil deve ter a coragem de tomar decisões que possam causar a rejeição de um projeto e sua refabricação. A equipe também deve ter humildade para reconhecer que os profissionais de tecnologia não possuem todas as respostas e que os experts em negócios e outros envolvidos devem ser respeitados e integrados ao processo.

Embora a AM sugira uma ampla gama de princípios de modelagem essenciais e suplementares, os que tornam a AM única são [Amb02a]:

Modele com um objetivo. O desenvolvedor que utilizar o AM deve ter um objetivo antes de criar o modelo (por exemplo, comunicar informações ao cliente ou ajudar a compreender melhor algum aspecto do software). Uma vez identificado o objetivo, ficará mais óbvio o tipo de notação a ser utilizado e o nível de detalhamento necessário.

Use modelos múltiplos. Há muitos modelos e notações diferentes que podem ser usados para descrever software. Somente um subconjunto é essencial para a maioria dos projetos. AM sugere que, para propiciar o insight necessário, cada modelo deve apresentar um aspecto diferente do sistema e somente aqueles que valorizem esses modelos para a audiência pretendida devem ser usados.

Viajar leve. Conforme o trabalho de engenharia de software prossegue, conserve apenas aqueles modelos que terão valor no longo prazo e despache o restante. Todo produto de trabalho mantido deve sofrer manutenção à medida que as mudanças ocorram. Isso representa trabalho que retarda a equipe. Ambler [Amb02a] observa que “Toda vez que se opta por manter um modelo, troca-se a agilidade pela conveniência de ter aquela informação acessível para a equipe de uma forma abstrata (já que, potencialmente, aumenta a comunicação dentro da equipe, assim como com os envolvidos no projeto)”.

Conteúdo é mais importante do que a representação. A modelagem deve transmitir informação para sua audiência pretendida. Um modelo sintaticamente perfeito que transmite pouco conteúdo útil não possui tanto valor como aquele com notações falhas que, no entanto, fornece conteúdo valioso para seu público-alvo.

Tenha conhecimento, domínio dos modelos e das ferramentas que for utilizar. Compreenda os pontos fortes e fracos de cada modelo e ferramenta usada para criá-lo.



“Viajar leve” é uma filosofia apropriada para todo o trabalho de engenharia de software. Construa apenas aqueles modelos que forneçam valor... Nem mais, nem menos.

Adapte localmente. A abordagem de modelagem deve ser adaptada às necessidades da equipe ágil.

Um segmento de vulto da comunidade da engenharia de software adotou a linguagem de modelagem unificada (Unified Modeling Language, UML)¹⁶ como o método preferido para análise representativa e para modelos de projeto. O Processo unificado (Capítulo 2) foi desenvolvido para fornecer uma metodologia para a aplicação da UML. Scott Ambler [Amb06] desenvolveu uma versão simplificada do UP que integra sua filosofia de modelagem ágil.

3.5.8 Processo Unificado Ágil (AUP)

O *processo unificado ágil* (Agile Unified Process) adota uma filosofia “serial para o que é amplo” e “iterativa para o que é particular” [Amb06] no desenvolvimento de sistemas computadorizados. Adotando as atividades em fases UP clássicas — *início, elaboração, construção e transição* —, AUP fornece uma camada serial (isto é, uma sequência linear de atividades de engenharia de software) que capacita uma equipe a visualizar o fluxo do processo geral de um projeto de software. Entretanto, dentro de cada atividade, a equipe itera ou se repete para alcançar a agilidade e para entregar incrementos de software significativos para os usuários finais tão rapidamente quanto possível. Cada iteração AUP dirige-se para as seguintes atividades [Amb06]:

- *Modelagem.* Representações UML do universo do negócio e do problema são criadas. Entretanto, para permanecer ágil, esses modelos devem ser “suficientemente bons e adequados” [Amb06] para possibilitar que a equipe prossiga.
- *Implementação.* Os modelos são traduzidos para o código-fonte.
- *Teste.* Como a XP, a equipe projeta e executa uma série de testes para descobrir erros e assegurar que o código-fonte se ajuste aos requisitos.
- *Aplicação.* Como a atividade de processo genérica discutida nos Capítulos 1 e 2, a aplicação neste contexto enfoca a entrega de um incremento de software e a aquisição de feedback dos usuários finais.
- *Configuração e gerenciamento de projeto.* No contexto da AUP, gerenciamento de configuração (Capítulo 22) refere-se a gerenciamento de alterações, de riscos e de controle de qualquer artefato¹⁷ persistente que sejam produzidos por uma equipe. Gerenciamento de projeto traciona e controla o progresso de uma equipe e coordena suas atividades.

FERRAMENTAS DO SOFTWARE



Engenharia de requisitos

Objetivo: O objetivo das ferramentas de desenvolvimento ágil é auxiliar em um ou mais aspectos do desenvolvimento ágil com ênfase em facilitar a geração rápida de software operacional. Essas ferramentas também podem ser usadas quando forem aplicados modelos de processos prescritivos (Capítulo 2).

Mecânica: A mecânica das ferramentas varia. Em geral, conjuntos de ferramentas ágeis englobam suporte automatizado para o planejamento de projetos, desenvolvimento de caso prático, coletânea de requisitos, projeto rápido, geração de código e teste.

Ferramentas representativas:¹⁸

Nota: Por ser o desenvolvimento ágil um tópico importante, a maioria dos vendedores de ferramentas de software

tende a vender ferramentas que dão suporte para a abordagem ágil. As ferramentas aqui observadas têm características que as tornam particularmente úteis para projetos ágeis.

OnTime, desenvolvida pela Axosoft (www.axosoft.com), fornece suporte para gerenciamento de processo ágil para uma variedade de atividades técnicas dentro do processo.

Ideogramic UML, desenvolvida pela Ideogramic (www.ideogramic.com), é um conjunto de ferramentas UML desenvolvido para uso em processo ágil.

Together Tool Set, distribuída pela Borland (www.borland.com), fornece uma mala de ferramentas que dão suporte para muitas atividades técnicas na XP e em outros processos ágeis.

¹⁶ Um breve tutorial sobre a UML é apresentado no Apêndice 1.

¹⁷ *Artefato persistente* é um modelo ou documento ou pacote de testes produzido pela equipe que será mantido por um período de tempo indeterminado. Não será descartado, uma vez que o incremento de software seja entregue.

¹⁸ Ferramentas observadas aqui não significam um aval, mas antes, uma amostra de ferramentas nesta categoria. Na maioria dos casos, os nomes das ferramentas são negociados por seus respectivos desenvolvedores.

- *Gerenciamento do ambiente.* Coordena a infraestrutura de processos que inclui padrões, ferramentas e outras tecnologias de suporte disponíveis para a equipe.

Embora o AUP possua conexões históricas e técnicas com a linguagem de modelagem unificada, é importante notar que a modelagem UML pode ser usado em conjunto com quaisquer modelos de processos ágeis descritos na Seção 3.5.

3.6 UM CONJUNTO DE FERRAMENTAS PARA O PROCESSO ÁGIL

PONTO-CHAVE

O “conjunto de ferramentas” que suporta os processos ágeis focaliza mais as questões pessoais do que as questões tecnológicas.

Alguns proponentes da filosofia ágil argumentam que as ferramentas de software automatizadas (por exemplo, ferramentas para projetos) deveriam ser vistas como um suplemento menor para as atividades, e não como pivô para o sucesso da equipe. Entretanto, Alistair Cockburn [Coc04] sugere que ferramentas podem gerar um benefício e que “equipes ágeis enfatizam o uso de ferramentas que permitam o fluxo rápido de compreensão. Algumas dessas ferramentas são sociais, iniciando-se até no estágio de contratação de pessoal. Algumas são tecnológicas, auxiliando equipes distribuídas a simular sua presença física. Muitas são físicas, permitindo sua manipulação em workshops”.

Pelo fato de que contratar as pessoas certas, ter a colaboração da equipe, manter a comunicação com os envolvidos e conseguir gerenciar de forma indireta constituírem elementos-chave em praticamente todos os modelos de processos ágeis, Cockburn afirma que “ferramentas” destinadas a esses itens são fatores críticos para a agilidade. Por exemplo, uma “ferramenta” alugada pode vir a ser um requisito para ter um membro de equipe de prospecção destinado a despendar algumas poucas horas em programação em dupla, com um membro já existente da equipe. O “encaixe” pode ser avaliado imediatamente.

“Ferramentas” voltadas para a comunicação e para a colaboração são, em geral, de tecnologia de base e incorporam qualquer mecanismo (“proximidade física, quadros brancos, papéis para pôster, fichas e lembretes adesivos” [Coc04]) que fornece informações e coordenação entre desenvolvedores. A comunicação ativa é obtida por meio de dinâmicas de grupo (por exemplo, programação em dupla), enquanto a comunicação passiva é obtida através dos “irradiadores de informações” (por exemplo, um *display* de um painel fixo que apresente o status geral dos diferentes componentes de um incremento). As ferramentas de gerenciamento de projeto não enfatizam tanto o diagrama de Gantt e o realoca com quadros de valores ganhos ou “gráficos de testes criados e cruzados com os anteriores... Outras ferramentas ágeis são utilizadas para otimizar o ambiente no qual a equipe ágil trabalha (por exemplo, mais áreas eficientes de encontro), também para ampliar a cultura da equipe por meio de incentivos para interações sociais (por exemplo, equipes alocadas juntas), para dispositivos físicos (por exemplo, lousas eletrônicas) e para ampliação (por exemplo, programação em dupla ou janela de tempo)” [Coc04].

Quaisquer dessas coisas são ferramentas? Serão, caso facilitem o trabalho desenvolvido por um membro da equipe ágil e venham a aprimorar a qualidade do produto final.

3.7 RESUMO

Em uma economia moderna, as condições de mercado mudam rapidamente, tanto o cliente quanto o usuário final devem evoluir e novos desafios competitivos surgem sem aviso. Os desenvolvedores têm de assumir uma abordagem de engenharia de software para permitir que permaneçam ágeis — definindo processos que sejam manipuláveis, adaptáveis, sem excessos, somente com o conteúdo essencial que possa adequar-se às necessidades do moderno mundo de negócios.

Uma filosofia ágil para a engenharia de software enfatiza quatro elementos-chave: a importância das equipes que se auto-organizam, que tenham controle sobre o trabalho por elas realizado, sobre a comunicação e sobre a colaboração entre os membros da equipe e entre os

desenvolvedores e seus clientes; o reconhecimento de que as mudanças representam oportunidades e ênfase na entrega rápida do software para satisfazer o cliente.

A Extreme Programming (XP) é o processo ágil mais amplamente utilizado. Organizada em quatro atividades metodológicas, planejamento, projeto, codificação e testes, a XP sugere um número de técnicas poderosas e inovadoras que possibilitam a uma equipe ágil criar versões de software frequentemente, propiciando recursos e funcionalidade estabelecidos anteriormente, e, então, priorizando os envolvidos.

Outros modelos de processos ágeis também enfatizam a colaboração humana e a auto-organização das equipes, mas definem suas próprias atividades metodológicas e selecionam diferentes pontos de ênfase. Por exemplo, ASD usa um processo iterativo que incorpora um planejamento cíclico iterativo, métodos de levantamento de requisitos relativamente rigorosos, e um ciclo de desenvolvimento iterativo que incorpora grupos focados nos clientes e revisões técnicas formais como mecanismos de feedback em tempo real. O Scrum enfatiza o uso de um conjunto de padrões de software que se mostrou efetivo para projetos com cronogramas apertados, requisitos mutáveis e aspectos críticos de negócio. Cada padrão de processo define um conjunto de tarefas de desenvolvimento e permite à equipe Scrum construir um processo que se adapta às necessidades do projeto. O método de desenvolvimento de sistemas dinâmicos (DSDM) defende o uso de um cronograma de tempos definidos (janela de tempo) e sugere que apenas o trabalho suficiente seja requisitado para cada incremento de software para facilitar o movimento ao incremento seguinte. Crystal é uma família de modelos de processos ágeis que podem ser desenvolvidos para uma característica específica de um projeto.

O desenvolvimento dirigido a funcionalidades (FDD) é ligeiramente mais “formal” que os outros métodos, mas ainda mantém agilidade ao focar a equipe do projeto no desenvolvimento de funcionalidades — validadas pelo cliente que possam ser implementadas em duas semanas ou menos. O desenvolvimento de software enxuto (LSD) adaptou os princípios de uma fabricação enxuta para o mundo da engenharia de software. A modelagem ágil (AM) afirma que modelagem é essencial para todos os sistemas, mas a complexidade, tipo e tamanhos de um modelo devem ser balizados pelo software a ser construído. O processo unificado ágil (AUP) adota a filosofia de “serial para o que é amplo” e “iterativa para o que é particular” para o desenvolvimento de software.

PROBLEMAS E PONTOS A PONDERAR

- 3.1.** Releia “The Manifesto for Agile Software Development” no início deste capítulo. Você consegue exemplificar uma situação em que um ou mais dos quatro “valores” poderiam levar a equipe a ter problemas?
- 3.2.** Descreva agilidade (para projetos de software) com suas próprias palavras.
- 3.3.** Por que um processo iterativo facilita o gerenciamento de mudanças? Todos os processos ágeis discutidos neste capítulo são iterativos? É possível completar um projeto com apenas uma iteração e ainda assim permanecer ágil? Justifique suas respostas.
- 3.4.** Cada um dos processos ágeis poderia ser descrito usando-se as atividades estruturais genéricas citadas no Capítulo 2? Construa uma tabela que associe as atividades genéricas às atividades definidas para cada processo ágil.
- 3.5.** Tente elaborar mais um “princípio de agilidade” que ajudaria uma equipe de engenharia de software a se tornar mais manobrável.
- 3.6.** Escolha um princípio de agilidade citado na Seção 3.3.1 e tente determinar se cada um dos modelos de processos apresentados neste capítulo demonstra o princípio. [Nota: Apresentei apenas uma visão geral desses modelos de processos; portanto, talvez não seja possível determinar se determinado princípio foi ou não tratado por um ou mais dos modelos, a menos que você pesquise mais a respeito (o que não é exigido para o presente problema).

- 3.7.** Por que os requisitos mudam tanto? Afinal de contas, as pessoas não sabem o que elas querem?
- 3.8.** A maior parte dos modelos de processos ágeis recomenda comunicação cara a cara. Mesmo assim, hoje em dia os membros de uma equipe de software e seus clientes podem estar geograficamente separados uns dos outros. Você acredita que isso implique que a separação geográfica seja algo a ser evitado? Você é capaz de imaginar maneiras para superar esse problema?
- 3.9.** Escreva uma história de usuário XP que descreva o recurso “sites favoritos” ou “bookmarks” disponível na maioria dos navegadores para Web.
- 3.10.** O que é uma solução pontual na XP?
- 3.11.** Descreva com suas próprias palavras os conceitos de refabricação e programação em dupla da XP.
- 3.12.** Leia um pouco mais a respeito e descreva o que é uma janela de tempo. Como isso ajuda uma equipe ASD na entrega de incrementos de software em um curto período de tempo?
- 3.13.** A regra dos 80% do DSDM e a abordagem de janelas de tempo definida para o ASD alcançam os mesmos resultados?
- 3.14.** Usando a planilha de padrões de processos apresentada no Capítulo 2, desenvolva um padrão de processo para qualquer um dos padrões Scrum da Seção 3.5.2.
- 3.15.** Por que o Crystal é considerado uma família de métodos ágeis?
- 3.16.** Usando o gabarito de recursos FDD descrito na Seção 3.5.5, defina um conjunto de recursos para um navegador Web. Agora, desenvolva vários recursos para o conjunto de recursos.
- 3.17.** Visite “The Official Agile Modeling Site” e faça uma lista completa de todos os princípios básicos e complementares do AM.
- 3.18.** O conjunto de ferramentas proposto na Seção 3.6 oferece suporte a muitos dos aspectos “menos prioritários” dos métodos ágeis. Como a comunicação é tão importante, recomende um conjunto de ferramentas real que poderia ser usado para melhorar a comunicação entre os interessados de uma equipe ágil.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A filosofia geral e os princípios subjacentes do desenvolvimento de software ágil são considerados em profundidade em muitos dos livros citados neste capítulo. Além destes, livros como os de Shaw e Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt (*Agile Software Building*, Springer, 2005) e Carmichael e Haywood (*Better Software Faster*, Prentice-Hall, 2002) trazem discussões interessantes sobre o tema. Aguanon (*Managing Agile Projects*, Multi-Media Publications, 2005), Highsmith (*Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004) e Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) apresentam uma visão geral sobre gerenciamento e consideram as questões envolvidas no gerenciamento de projetos. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) retrata uma pesquisa de princípios, processos e práticas ágeis. Uma discussão que vale a pena sobre o delicado equilíbrio entre agilidade e disciplina é fornecida por Booch e seus colegas (*Balancing Agility and Discipline*, Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, 2009) enumera os princípios, padrões e práticas necessários para desenvolver “código limpo” em um ambiente de engenharia de software ágil. Leffingwell (*Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) discute estratégias para dar maior corpo às práticas ágeis para poderem ser usadas em grandes projetos. Lippert e Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley, 2006) discutem o uso da refabricação quando aplicada a sistemas grandes e complexos.

Stamelos e Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) trazem técnicas SQA que estão em conformidade com a filosofia ágil.

Foram escritos dezenas de livros sobre Extreme Programming ao longo da última década. Beck (*Extreme Programming Explained: Embrace Change*, 2. ed., Addison-Wesley, 2004) ainda é o tratado de maior autoridade sobre o tema. Além desse, Jeffries e seus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi e Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk e Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001), bem como Auer e seus colegas (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001), fornecem uma discussão básica da XP juntamente com uma orientação sobre como melhor aplicá-la. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) adota uma visão crítica em relação à XP, definindo quando e onde ela é apropriada. Uma análise aprofundada da programação em dupla é apresentada por McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

A ASD é tratada em profundidade por Highsmith [Hig00]. Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) discute o uso do Scrum para projetos que possuem um grande impacto sobre as empresas. Os detalhes práticos do Scrum são debatidos por Schwaber e Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Tratados úteis sobre o DSDM foram escritos pelo DSDM Consortium (*DSDM: Business Focused Development*, 2. ed., Pearson Education, 2003) e Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Cockburn (*Crystal Clear*, Addison-Wesley, 2005) traz uma excelente visão geral da família Crystal de processos. Palmer e Felsing [Pal02] apresentam um tratado detalhado acerca do FDD. Carmichael e Haywood (*Better Software Faster*, Prentice-Hall, 2002) é mais um tratado útil sobre o FDD que inclui uma jornada passo a passo através da mecânica do processo. Poppendieck e Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) dão diretrizes para gerenciar e controlar projetos ágeis. Ambler e Jeffries (*Agile Modeling*, Wiley, 2002) discutem a AM com certa profundidade.

Uma grande variedade de fontes de informação sobre desenvolvimento de software ágil está disponível na Internet. Uma lista atualizada de referências na Web relevantes ao processo ágil pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

Nesta parte do livro *Engenharia de software: uma abordagem profissional*, você aprenderá os princípios, conceitos e métodos usados para criar modelos de necessidades e de projeto de alta qualidade.

As seguintes questões são tratadas nos próximos capítulos:

- Quais conceitos e princípios orientam a prática da engenharia de software?
- O que é engenharia de necessidades e quais os conceitos subjacentes que levam a uma análise de necessidades adequada?
- Como é criado o modelo de necessidades e quais seus elementos?
- Quais os elementos de um bom projeto?
- Como o desenho da arquitetura estabelece uma estrutura para todas as demais ações de projeto e quais os modelos utilizados?
- Como projetar componentes de software de alta qualidade?
- Que conceitos, modelos e métodos são aplicados quando é projetada uma interface para o usuário?
- O que é projeto baseado em padrões?
- Que estratégias e métodos especializados são usados para projetar WebApps?

Assim que essas perguntas forem respondidas, você estará mais bem preparado para a prática da engenharia de software.

PRINCÍPIOS QUE ORIENTAM A PRÁTICA

CONCEITOS-CHAVE

princípios fundamentais	109
princípios que governam a codificação	120
a comunicação	112
a disponibilização	122
a modelagem ...	116
o planejamento ..	114
o projeto	118
os requisitos	117
os testes	121

Em um livro que pesquisa o cotidiano e reflexões dos engenheiros de software, Ellen Ullman [Ull 97] descreve um pouco da experiência de vida, à medida que relata os pensamentos do profissional atuante sob pressão:

Não tenho ideia de que horas são. Não há janelas neste escritório, nem relógio, somente o LED vermelho de um micro-ondas piscando 12:00, 12:00, 12:00, 12:00. Joel e eu estamos programando há dias. Temos um defeito, um demônio teimoso... Da mesma forma, o pisca vermelho não se acerta em temporização, como se fosse uma representação de nossos cérebros, que, de alguma forma, acabaram sincronizados com a mesma faísca temporal do LED...

Em que estamos trabalhando?... Os detalhes me escapam agora. Podemos estar auxiliando os pobres ou sintonizando uma série de rotinas de baixo nível, a fim de verificar os bits de um protocolo de distribuição de dados. Nem me importo. Deveria me importar; em uma outra época — talvez, quando sairmos desta sala cheia de computadores —, me importarei muitíssimo com o porquê, para quem e para qual finalidade estou desenvolvendo software. Mas neste exato momento: não. Ultrapassei uma camada na qual o mundo real e seus usuários não mais importam. Sou um engenheiro de software...

Muitos leitores estarão capacitados para lidar com esse lado sombrio da engenharia de software, com ponderação.

PANORAMA

O que é? A prática da engenharia de software consiste em uma série de princípios, conceitos, métodos e ferramentas que devem ser considerados ao planejar e desenvolver um software. Princípios que direcionam a ação estabelecem a infraestrutura a partir da qual a engenharia de software é conduzida.

Quem realiza? Praticantes (engenheiros de software) e seus coordenadores (gerentes) desenvolvem uma variedade de tarefas de engenharia de software.

Por que é importante? O processo de software propicia a todos os envolvidos na criação de um sistema computacional ou produto para computador um roteiro para conseguir chegar a um destino com sucesso. A prática fornece o detalhamento necessário para seguir ao longo da estrada. Aponta para onde estão as pontes, os conjuntos de rodovias, as bifurcações. Auxilia a compreender os conceitos e princípios que devem ser compreendidos e seguidos para que se dirija com rapidez e segurança. Orienta quanto ao como seguir, onde diminuir e aumentar o ritmo. No contexto da engenharia de software, a prática

consiste no que se realiza diariamente, à medida que o software evolui de ideia a realidade.

Quais são as etapas envolvidas? Aplicam-se três elementos práticos, independentemente do modelo de processo escolhido. São eles: princípios, conceitos e métodos. Um quarto elemento de prática, ferramentas, sustenta a aplicação dos métodos.

Qual é o artefato? A prática engloba as atividades técnicas que produzem todos os artefatos definidos pelo modelo de processo de software escolhido.

Como garantir que o trabalho foi realizado corretamente? Primeiro, compreenda completamente os princípios aplicados ao trabalho (por exemplo, projeto) que está sendo realizado no momento. Em seguida, esteja certo de que escolheu um método apropriado, certifique-se de ter compreendido como aplicar o método, use ferramentas automatizadas, quando estas forem apropriadas à tarefa, e seja inflexível quanto à necessidade de técnicas para se assegurar da qualidade dos produtos de trabalho produzidos.

As pessoas que criam software praticam a arte ou ofício ou disciplina¹ em que consiste a engenharia de software. Mas em que consiste a “prática” de engenharia de software? De forma genérica, *prática* é um conjunto de conceitos, princípios, métodos e ferramentas aos quais um engenheiro de software recorre diariamente. A prática permite aos coordenadores (gerentes) administrar os projetos e aos engenheiros de software criar programas computacionais. A prática preenche um modelo de processo de software com recursos de como fazer para ter o trabalho realizado em termos de gerenciamento e de tecnologia. Transforma uma abordagem desfocada e confusa em algo mais organizado, mais efetivo e mais propenso a obter sucesso.

Diversos aspectos da engenharia de software serão examinados ao longo do livro.

Neste capítulo, o foco será em princípios e conceitos que norteiam a prática da engenharia de software em geral.

4.1 CONHECIMENTO DA ENGENHARIA DE SOFTWARE

Em um editorial publicado na *IEEE Software* uma década atrás, Steve McConnell [McC99] fez o seguinte comentário:

Muitos desenvolvedores (de software) veem o conhecimento da engenharia de software quase que exclusivamente como um conhecimento de tecnologias específicas: Java, Perl, html, C++, Linux, Windows NT etc. O conhecimento de detalhes específicos de tecnologia é necessário para desenvolver a programação de computador. Se alguém o contrata para desenvolver um programa em C++, você deverá saber algo sobre C++ para fazer seu programa funcionar.

Frequentemente, diz-se que conhecimento sobre desenvolvimento de software tem uns três anos de vida: metade do que se precisa saber hoje estará obsoleto daqui a três anos. No que diz respeito ao conhecimento de tecnologia, essa afirmação está próxima da verdade. Mas há um outro tipo de conhecimento sobre desenvolvimento de software — um tipo visto como “princípios da engenharia de software” — que não possui três anos de meia-vida. Tais princípios provavelmente servirão ao programador profissional durante toda a sua carreira.

McConnell continua afirmando que a base do conhecimento em engenharia de software (por volta do ano 2000) evoluiu para uma “essência estável” que ele estimou representar cerca de “75% do conhecimento necessário para desenvolver um sistema complexo”. No entanto, no que consiste essa essência estável?

Como indica McConnell, princípios essenciais — ideias elementares que guiam engenheiros de software em seus trabalhos — fornecem, em nossos dias, uma infraestrutura a partir da qual os modelos de engenharia de software, métodos e ferramentas podem ser aplicados e avaliados.

4.2 PRINCÍPIOS FUNDAMENTAIS

“Teoricamente, não há nenhuma diferença entre a teoria e a prática. Porém, na prática, ela existe.”

Jan van de Snepscheut

A engenharia de software é norteada por um conjunto de princípios fundamentais que auxiliam na aplicação de um processo de software significativo e na execução de métodos de engenharia de software efetivos. No nível relativo a processo, os princípios fundamentais estabelecem uma infraestrutura filosófica que guia uma equipe de software à medida que desenvolve atividades de apoio e estruturais, navega no fluxo do processo e cria um conjunto de artefatos de engenharia de software. Quanto à prática, os princípios fundamentais estabelecem um artefato e regras que servem como guias conforme se analisa um problema, projeta uma solução, implementa e testa uma solução e, por fim, emprega o software na comunidade de usuários.

No Capítulo 1, identificou-se uma série de princípios fundamentais que abrange o processo e a prática da engenharia de software: (1) fornecer valor aos usuários finais, (2) simplificar, (3)

¹ Certos autores argumentam que um desses termos exclui outros. Na realidade, a engenharia de software se aplica a todos os três.

manter a visão (do produto e do projeto), (4) reconhecer que outros usuários consomem (e devem entender) o que se produz, (5) manter abertura para o futuro, (6) planejar antecipadamente para o reuso, e (7) raciocinar!

Embora tais princípios gerais sejam importantes, são caracterizados em um nível tão elevado de abstração que, às vezes, torna-se difícil a tradução para a prática diária da engenharia de software. Nas subseções seguintes, há um maior detalhamento dos princípios essenciais que conduzem o processo e a prática.

4.2.1 Princípios que orientam o processo

Na Parte 1 deste livro, discutiu-se a importância do processo de software e descreveram-se os diferentes modelos de processos propostos para o trabalho de engenharia de software. Não importando se um modelo é linear ou iterativo, prescritivo ou ágil, pode ser caracterizado usando-se uma metodologia de processo genérica aplicável a todos os modelos de processo. O conjunto de princípios fundamentais apresentados a seguir pode ser aplicado à metodologia e, por extensão, a todos os processos de software.

Princípio 1. Seja ágil. Não importa se o modelo de processo que você escolheu é prescritivo ou ágil, os princípios básicos do desenvolvimento ágil devem comandar sua abordagem. Todo aspecto do trabalho deve enfatizar a economia de ações — mantenha a abordagem técnica tão simples quanto possível, mantenha os produtos tão concisos quanto possível e tome decisões localmente sempre que possível.

Princípio 2. Concentre-se na qualidade em todas as etapas. A condição final para toda atividade, ação e tarefa do processo deve focar na qualidade do produto produzido.

Princípio 3. Esteja pronto para adaptações. Processo não é uma experiência religiosa e não há espaço para dogma. Quando necessário, adapte sua abordagem às restrições impostas pelo problema, pelas pessoas e pelo próprio projeto.

Princípio 4. Monte uma equipe efetiva. O processo e a prática de engenharia de software são importantes, mas o fator mais importante são as pessoas. Forme uma equipe que se auto-organize, que tenha confiança e respeito mútuos.

Princípio 5. Estabeleça mecanismos para comunicação e coordenação. Os projetos falham devido à omissão de informações importantes nas “fendas” da estrutura do programa e/ou devido a indivíduos que falham na coordenação de seus esforços para criar um produto final bem-sucedido. Esses são itens de gerenciamento e devem ser direcionados.

Princípio 6. Gerencie mudanças. A abordagem pode ser tanto formal quanto informal, entretanto os mecanismos devem ser estabelecidos para gerenciar a maneira como as mudanças serão solicitadas, avaliadas, aprovadas e implementadas.

Princípio 7. Avalie os riscos. Uma série de coisas pode dar errada quando um software é desenvolvido. É essencial estabelecer planos de contingência.

Princípio 8. Gere artefatos que forneçam valor para outros. Crie apenas artefatos que proporcionarão valor para outro processo, atividades, ações ou tarefas. Todo artefato produzido como parte da prática da engenharia de software será repassado para alguém mais. Uma lista de funções e características requisitadas será repassada para a pessoa (ou pessoas) que irá desenvolver um projeto, o projeto será repassado para aqueles que gerarão o código e assim por diante. Certifique-se de que o artefato contenha a informação necessária sem ambiguidades ou omissões.

A Parte 4 deste livro enfoca o projeto, os fatores de gerenciamento do processo e aborda os aspectos variados de cada um desses princípios com certo detalhe.



Todo projeto e toda equipe são únicos. Isso significa que se deve adaptar o processo para que melhor se ajuste às suas necessidades.

“A verdade da questão é que sempre sabemos a coisa certa a ser feita. A parte difícil é fazê-la.”

General H. Norman Schwarzkopf

4.2.2 Princípios que orientam a prática

A prática da engenharia de software tem um objetivo primordial único — entregar dentro do prazo, com alta qualidade, o software operacional contendo funções e características que satisfaçam as necessidades de todos os envolvidos. Para atingir esse objetivo, deve-se adotar um conjunto de princípios fundamentais que orientem o trabalho técnico. Tais princípios são importantes, independentemente do método de análise ou projeto aplicado, das técnicas de desenvolvimento (por exemplo, linguagem de programação, ferramentas de automação) escolhidas, ou da abordagem de verificação e validação utilizada. O cumprimento de princípios fundamentais a seguir é essencial para a prática de engenharia de software:

PONTO-CHAVE

Os problemas tornam-se mais fáceis para resolver quando subdivididos por interesses, cada um distinto, individualmente solucionável e passível de verificação.

Princípio 1. Divida e conquiste. De forma mais técnica, a análise e o projeto sempre devem enfatizar a *separação por interesses*² (*— separation of concerns — SoC*). Um problema terá sua resolução mais fácil se for subdividido em conjuntos de interesses. Na forma ideal, cada interesse fornece uma funcionalidade distinta a ser desenvolvida, e, em alguns casos, validada, independentemente de outros negócios.

Princípio 2. Compreenda o uso da abstração. Na essência, abstrair é simplificar algum elemento complexo de um sistema utilizado para comunicar significado em uma única frase. Quando se usa a abstração *planilha*, assume-se que se compreende o que vem a ser uma planilha, a estrutura geral de conteúdo que uma planilha apresenta e as funções típicas que podem ser aplicadas a ela. Na prática de engenharia de software, usam-se muitos níveis diferentes de abstração, cada um incorporando ou implicando um significado que deve ser comunicado. No trabalho de análise de projeto, uma equipe de software normalmente inicia com modelos que representam altos níveis de abstração (por exemplo, planilha) e, aos poucos, refina tais modelos em níveis de abstração mais baixos (por exemplo, uma coluna ou uma função de soma).

Joel Spolsky [Spo02] sugere que “todas as abstrações não triviais são, até certo ponto, frágeis”. O objetivo de uma abstração é eliminar a necessidade de comunicar detalhes. Algumas vezes, efeitos problemáticos se precipitam em virtude da “aresta” dos detalhes. Sem a compreensão dos detalhes, a causa de um problema não poderá ser facilmente diagnosticada.

Princípio 3. Esforce-se por consistência. Seja criando um modelo de requisitos, desenvolvendo um projeto de software, gerando código-fonte, seja criando pacotes de teste, o princípio de consistência sugere que um contexto familiar facilita o uso do software. Consideremos, por exemplo, o projeto de uma interface para o usuário de uma WebApp. A colocação consistente de menu de opções, o uso consistente de um esquema de cores e de ícones identificáveis, tudo colabora para uma interface ergonomicamente forte.

Princípio 4. Foque na transferência de informação. Software trata a transferência de informações: do banco de dados para um usuário final, de um sistema judiciário para uma aplicação na Web (WebApp), do usuário final para uma interface gráfica (GUI), de um sistema operacional de um componente de software para outro; a lista é quase infinita. Em todos os casos, a informação flui através de uma interface, e, como consequência, há a possibilidade de erros, omissões e ambiguidade. A implicação desse princípio é que se deve prestar especial atenção para interfaces de análise, projeto, construção e testes.

Princípio 5. Construa software que apresente modularidade efetiva. A separação por interesse (Princípio 1) estabelece uma filosofia para software. A *modularidade* fornece um mecanismo para colocar a filosofia em prática. Qualquer sistema complexo pode ser dividido em módulos (componentes), porém a boa prática de engenharia de software demanda mais

2 N. de T.: A palavra *concern* foi traduzida como interesse ou afinidade. Dividir é uma técnica utilizada para lidar com complexidade. Uma das estratégias para dividir é separar usando critérios como interesses ou afinidade tanto, no domínio do problema como no domínio da tecnologia do software.



Use padrões (Capítulo 12) para armazenar conhecimento e experiência para as futuras gerações de engenheiros de software.

que isso. A modularidade deve ser *efetiva*. Isto é, cada módulo deve focalizar exclusivamente um aspecto bem restrito do sistema — deve ser coeso em sua função e/ou apresentar conteúdo bem preciso. Além disso, os módulos devem ser interconectados de uma maneira relativamente simples — cada módulo deve apresentar baixo acoplamento com outros módulos, fontes de dados e outros aspectos ambientais.

Princípio 6. Padronize. Brad Appleton [App00] propõe que:

O objetivo da padronização é criar uma fonte literária para ajudar os desenvolvedores de software a solucionar problemas recorrentes, encontrados ao longo de todo o desenvolvimento de software. Os padrões ajudam a criar uma linguagem compartilhada para transmitir conteúdo e experiência acerca dos problemas e de suas soluções. Codificar formalmente tais soluções e suas relações permite que se armazene com sucesso a base do conhecimento, a qual define nossa compreensão sobre boas arquiteturas, correspondendo às necessidades de seus usuários.

Princípio 7. Quando possível, represente o problema e sua solução sob uma série de perspectivas diferentes. Ao analisar um problema e sua solução sob uma série de perspectivas diferentes é mais provável que se obtenha uma melhor visão e que os erros e omissões sejam revelados. Por exemplo, um modelo de requisitos pode ser representado usando um ponto de vista orientado a dados, um ponto de vista orientado a funções ou um ponto de vista comportamental (Capítulos 6 e 7). Cada um deles fornece uma visão diferente do problema e de seus requisitos.

Princípio 8. Lembre-se de que alguém fará a manutenção do software. A longo prazo, conforme defeitos são descobertos, o software será corrigido, adaptado de acordo com as mudanças de seu ambiente e ampliado conforme solicitação por mais capacidade dos envolvidos. As atividades de manutenção podem ser facilitadas se for aplicada uma prática de engenharia de software consistente ao longo do processo.

Esses princípios não constituem o todo necessário para a construção de um software de alta qualidade, mas realmente estabelecem uma base para todo método de engenharia de software discutido neste livro.

4.3 PRINCÍPIOS DAS ATIVIDADES METODOLÓGICAS

“O engenheiro ideal é uma composição: não é um cientista, não é um matemático, não é um sociólogo ou escritor; mas pode utilizar o conhecimento e as técnicas de qualquer uma ou de todas as disciplinas para resolver os problemas de engenharia.”

N. W. Dougherty

Nas seções seguintes, serão apresentados princípios que constituem uma forte base para o sucesso de cada atividade metodológica genérica, definida como parte do processo de software. Em muitos casos, são um aprimoramento dos princípios apresentados na Seção 4.2, fundamentais e simplificados, situando-se em um nível de abstração mais baixo.

4.3.1 Princípios da comunicação

Antes que os requisitos dos clientes sejam analisados, modelados ou especificados, eles devem ser coletados através da atividade de comunicação. Um cliente apresenta um problema que pode ser amenizado por uma solução baseada em computador. Responde-se ao pedido de ajuda e inicia-se a comunicação. Entretanto, o percurso da comunicação ao entendimento é, frequentemente, acidentado.

A comunicação efetiva (entre parceiros técnicos com o cliente, com outros parceiros interessados e com gerenciadores de projetos) constitui uma das atividades mais desafiadoras. Nesse contexto, discutem-se princípios de comunicação aplicados na comunicação com o cliente. Entretanto, muitos se aplicam a todas as formas de comunicação.

Princípio 1. Ouça. Concentre-se mais em ouvir do que em se preocupar com respostas. Solicite esclarecimento caso necessário, e evite interrupções constantes. Nunca se mostre contestador tanto em palavras quanto em ações (por exemplo, revirar olhos e menear a cabeça) enquanto uma pessoa estiver falando.



Antes de se comunicar, assegure-se de compreender o ponto de vista alheio, suas necessidades, e saiba ouvir.

"Planeje perguntas e respostas, trace o caminho mais curto para a maioria das perplexidades."

Mark Twain

Princípio 2. *Prepare-se antes de se comunicar.* Dedique tempo para compreender o problema antes de se encontrar com outras pessoas. Se necessário, realize algumas pesquisas para entender o jargão da área de negócios em questão. Caso seja sua a responsabilidade pela condução de uma reunião, prepare, com antecedência, uma agenda.

Princípio 3. *Alguém deve facilitar a atividade.* Toda reunião de comunicação deve ter um líder (um facilitador) para manter a conversação direcionada e produtiva, (2) para mediar qualquer conflito que ocorra e (3) para garantir que outros princípios sejam seguidos.

Princípio 4. *Comunique-se pessoalmente.* No entanto, em geral, comunicar-se pessoalmente é mais produtivo tendo um outro representante presente. Por exemplo, um participante pode criar um desenho ou um esboço de documento que servirá como foco para a discussão.

Princípio 5. *Anote e documente as decisões.* As coisas tendem a cair no esquecimento. Algum participante da comunicação deve servir como um "gravador" e anotar todos os pontos e decisões importantes.

Princípio 6. *Esforce-se por colaboração.* Colaboração e consenso ocorrem quando o conhecimento coletivo dos membros da equipe for usado para descrever funções e características do produto ou sistema. Cada pequena colaboração servirá para estabelecer confiança entre os membros e chegar a um objetivo comum.

Princípio 7. *Mantenha o foco e crie módulos para a discussão.* Quanto mais pessoas envolvidas, maior a probabilidade de a discussão seguir de um tópico a outro. O facilitador deve manter a conversa modular, abandonando um tópico somente depois de este ter sido resolvido (veja, entretanto, o Princípio 9).

Princípio 8. *Faltando clareza, represente graficamente.* A comunicação verbal flui até certo ponto. Um esboço ou uma representação gráfica pode permitir maior clareza quando palavras são insuficientes.

Princípio 9. (a) *Uma vez de acordo, siga em frente.* (b) *Se não chegar a um acordo, siga em frente.* (c) *Se uma característica ou função estiver obscura e não puder ser elucidada no momento, siga em frente.* A comunicação, assim como qualquer outra atividade da engenharia de software, toma tempo. Em vez de ficar interagindo indefinidamente, os participantes precisam reconhecer que muitos tópicos exigem discussão (veja o Princípio 2) e que "seguir em frente" é, algumas vezes, a melhor maneira de alcançar agilidade na comunicação.

Princípio 10. *Negociação não é uma contestação nem um jogo. Funciona melhor quando ambas as partes saem ganhando.* Há muitas ocasiões em que se têm de negociar funções e características, prioridades e prazos de entrega. Se a equipe interagiu adequadamente, todas as partes têm um objetivo comum. Mesmo assim, a negociação exigirá compromisso de todos.

? O que Acontecerá caso não consiga chegar a um acordo com um cliente sobre alguma questão relativa ao projeto?



A diferença entre clientes e usuários finais

Os engenheiros de software se comunicam com muitos interessados diferentes, mas clientes e usuários finais têm o maior impacto sobre o trabalho técnico que virá a seguir. Em alguns casos, cliente e usuário final são a mesma pessoa, mas, para muitos projetos, são pessoas que trabalham para gerentes diferentes, em organizações diferentes.

Cliente é a pessoa ou grupo que: (1) originalmente, requisita o software a ser construído, (2) define os objetivos gerais do negócio para o software, (3) fornece os requisitos básicos

do produto e (4) coordena os recursos financeiros para o projeto. Em uma negociação de sistemas ou de produtos, o cliente, com frequência, é o departamento de marketing. Em um ambiente de tecnologia da informação (TI), o cliente pode ser um departamento ou componente de negócio.

Usuário final é uma pessoa ou grupo que (1) irá realmente usar o software que é construído para atingir algum propósito de negócio e (2) irá definir os detalhes operacionais do software de modo que o objetivo seja alcançado.

INFORMAÇÕES

CASA SEGURA



Erros de comunicação

Cena: Área de trabalho da equipe de engenharia de software

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software.

Conversa:

Ed: "O que você ouviu falar sobre o projeto CasaSegura?"

Vinod: "A reunião inicial está marcada para a próxima semana."

Jamie: "Eu já andei investigando, mas não deu muito certo."

Ed: "O que você quer dizer com isso?"

Jamie: "Bem, dei uma ligada para Lisa Perez. Ela é a mandachuva de marketing dessa coisa."

Vinod: "E...?"

Jamie: "Eu queria que ela me desse informações sobre as características e funções do CasaSegura... esse tipo de coisa."

Mas, em vez disso, ela começou a me fazer perguntas sobre sistemas de segurança, sistemas de vigilância... Eu não sou um especialista na área."

Vinod: "O que isso lhe diz?" (Jamie encolhe os ombros.)

Vinod: "Isso quer dizer que o marketing precisará de nós como consultores e que é melhor nos prepararmos nesta área de produto antes da primeira reunião. Doug disse que quer que 'colaboremos' com nosso cliente; portanto, é melhor aprendermos como fazer isso."

Ed: "Provavelmente teria sido melhor ter dado uma passada na sala dela. Telefonemas simplesmente não funcionam bem para esse tipo de coisa."

Jamie: "Vocês dois estão certos. Temos que agir em conjunto ou nossos primeiros contatos serão difíceis."

Vinod: "Vi o Doug lendo um livro sobre 'engenharia de requisitos'. Aposto que ele enumera alguns princípios de comunicação efetiva. Vou pedi-lo emprestado amanhã mesmo."

Jamie: "Boa ideia... depois você poderá nos ensinar."

Vinod (sorrindo): "É isso aí."

4.3.2 Princípios de planejamento

A atividade de comunicação ajuda a definir metas gerais e objetivos (sujeitos a mudanças, é claro, à medida que o tempo passa). Entretanto, compreender essas metas e objetivos não é o mesmo que definir um plano para alcançá-los. A atividade de planejamento engloba um conjunto de práticas técnicas e gerenciais que permitem à equipe de software definir um roteiro à medida que segue na direção de seus objetivos estratégicos e táticos.

Por mais que se tente, é impossível prever com exatidão como um projeto de software irá evoluir. Não há nenhuma maneira simples para determinar quais problemas técnicos não previstos serão encontrados, quais enganos ocorrerão ou quais itens de negócio sofrerão mudanças. Ainda assim, uma boa equipe deve planejar sua abordagem.

Há muitas filosofias de planejamento.³ Algumas pessoas são "minimalistas", afirmando que alterações, frequentemente, evidenciam a necessidade de um plano detalhado. Outras são "tradicionalistas", afirmando que o plano fornece um roteiro efetivo e que, quanto mais detalhes apresentar, menos probabilidade terá a equipe de se perder. Outras ainda são "agilistas", afirmando que um rápido "planejamento do jogo" pode ser necessário, entretanto, o roteiro surgirá como "verdadeiro trabalho" no início do software.

O que fazer? Em muitos projetos, planejamento em excesso representa consumo de tempo sem resultado produtivo (mudanças demais), entretanto, pouco planejamento é uma receita para o caos. Como a maioria das coisas na vida, o planejamento deveria ser conduzido de forma moderada, o suficiente para servir de guia para a equipe – nem mais, nem menos. Não importando o rigor com o qual o planejamento seja feito, os seguintes princípios sempre se aplicam:

Princípio 1. Compreenda o escopo do projeto. É impossível usar um roteiro caso não se saiba para onde se está indo. O escopo indica ao grupo um destino.

Princípio 2. Envolver os interessados na atividade de planejamento. Os interessados definem prioridades e estabelecem as restrições de projeto. Para adequar essas característi-

"Na preparação para a batalha sempre achei que os planos fossem inúteis, mas planejamento é indispensável."

General Dwight D. Eisenhower

WebRef

Um excelente banco de informações sobre planejamento e gerenciamento de projetos pode ser encontrado em www.4pm.com/repository.htm.

3 Na Parte 4 do livro é apresentada uma discussão detalhada sobre planejamento e gerenciamento de projetos de software.

cas, os engenheiros devem negociar com frequência a programação de entrega, cronograma e outras questões relativas ao projeto.

Princípio 3. Reconheça que o planejamento é iterativo. Um plano de projeto jamais é gravado na pedra. Conforme se inicia o trabalho, muito provavelmente ocorrerão alterações. Como consequência, o plano deverá ser ajustado para incluir as alterações. Além do mais, os modelos de processos incremental e iterativo exigem replanejamento após a entrega de cada incremento de software baseado em feedbacks recebidos dos usuários.

Princípio 4. Faça estimativas com base no que conhece. O objetivo da estimativa é oferecer indicações de esforço, custo e prazo para a realização, com base na compreensão atual do trabalho a realizar. Se a informação for vaga ou não confiável, as estimativas serão igualmente não confiáveis.

Princípio 5. Considere os riscos ao definir o plano. Caso tenha identificado riscos de alto impacto e alta probabilidade, o planejamento de contingência será necessário. Além disso, o plano de projeto (inclusive o cronograma) deve ser ajustado para incluir a tendência de um ou mais desses riscos ocorrerem.

Princípio 6. Seja realista. As pessoas não trabalham 100% de todos os dias. Sempre há interferência de ruído em qualquer comunicação humana. Omissões e ambiguidades são fatos da vida. Mudanças ocorrem. Até mesmo os melhores engenheiros de software cometem erros. Essas e outras realidades devem ser consideradas quando se estabelece um plano de projeto.

Princípio 7. Ajuste particularidades ao definir o plano. *Particularidades* referem-se ao nível de detalhamento introduzido conforme o plano de projeto é desenvolvido. Um plano com alto grau de particularidade fornece considerável detalhamento de tarefas planejadas para incrementos em intervalos relativamente curtos para que o rastreamento e controle ocorram com frequência. Um plano com baixo grau de particularidade resulta em tarefas mais amplas para intervalos maiores. Em geral, particularidades variam de alto para baixo conforme o cronograma de projeto se distancia da data corrente. Nas semanas ou meses seguintes, o projeto pode ser planejado com detalhes significativos. As atividades que não serão realizadas por muitos meses não requerem alto grau de particularidade (muito pode ser alterado).

Princípio 8. Defina como se pretende garantir a qualidade. O plano deve determinar como a equipe pretende garantir a qualidade. Se forem necessárias revisões técnicas⁴, deve-se agendá-las. Se a programação pareada for utilizada (Capítulo 3), deve-se definir claramente dentro do plano.

Princípio 9. Descreva como pretende adequar as alterações. Mesmo o melhor planejamento pode ser prejudicado por alteração sem controle. Deve-se identificar como as alterações serão integradas ao longo do trabalho em engenharia. Por exemplo, o cliente pode solicitar uma alteração a qualquer momento? Se for solicitada uma mudança, a equipe é obrigada a implementá-la imediatamente? Como é avaliado o impacto e o custo de uma alteração?

Princípio 10. Verifique o plano frequentemente e faça ajustes necessários. Os projetos de software atrasam uma vez ou outra. Portanto, é de bom senso checar diariamente seu progresso, procurando por áreas ou situações problemáticas nas quais o que foi programado não está em conformidade com o trabalho real a ser conduzido. Ao surgir um descompasso, deve-se ajustar o plano adequadamente. Para máxima eficiência, todos da equipe de software devem participar da atividade de planejamento. Só dessa maneira os membros estarão comprometidos (engajados) com o plano.

"O sucesso deve-se mais ao bom senso do que à genialidade."

An Wang

PONTO-CHAVE

O termo particularidade refere-se a detalhes por meio dos quais alguns elementos do planejamento são representados ou conduzidos.

4 As revisões técnicas são discutidas no Capítulo 15.

4.3.3 Princípios de modelagem

Criam-se modelos para uma melhor compreensão do que será realmente construído. Quando a entidade for algo físico (por exemplo, um edifício, um avião, uma máquina), pode-se construir um modelo que seja idêntico na forma e no formato, porém menor em escala. Entretanto, quando a entidade a ser construída for software, nosso modelo deve assumir uma forma diferente. Deve ser capaz de representar as informações que o software transforma, a arquitetura e as funções que possibilitam a transformação, as características que os usuários desejam e o comportamento do sistema à medida que a transformação ocorra. Os modelos devem cumprir esses objetivos em diferentes níveis de abstração — primeiro, descrevendo o software do ponto de vista do cliente e, posteriormente, em um nível mais técnico.

PONTO-CHAVE

Os modelos de requisitos representam os requisitos dos clientes. Os modelos de projeto oferecem uma especificação concreta para a construção do software.

No trabalho de engenharia de software, podem ser criadas duas classes de modelos: modelos de requisitos e modelos de projeto. Os *modelos de requisitos* (também denominados *modelos de análise*) representam os requisitos dos clientes, descrevendo o software em três domínios diferentes: o domínio da informação, o domínio funcional e o domínio comportamental. Os *modelos de projeto* representam características do software que ajudam os desenvolvedores a construí-lo efetivamente: a arquitetura, a interface para o usuário e os detalhes quanto a componentes.

Em seu livro sobre modelagem ágil, Scott Ambler e Ron Jeffries [Amb02b] estabelecem uma série de princípios⁵ de modelagem destinados àqueles que usam o modelo de processos ágeis (Capítulo 3), mas são apropriados para todos os engenheiros de software que executam ações e tarefas de modelagem:

Princípio 1. O objetivo principal da equipe de software é construir software, e não criar modelos. Agilidade significa entregar software ao cliente no menor prazo possível. Os modelos que fazem com que isso aconteça são criações valiosas, entretanto, os que retardam o processo oferecem pouca novidade, devem ser evitados.

Princípio 2. Viaje leve — não crie mais modelos do que necessita. Todo modelo criado deve ser atualizado quando ocorrem alterações. E, mais importante, todo modelo novo demanda tempo que poderia ser dispendido em construção (codificação e testes). Portanto, crie somente aqueles modelos que facilitem mais e diminuam o tempo para a construção do software.

Princípio 3. Esforce-se ao máximo para produzir o modelo mais simples possível. Não exagere no software [Amb02b]. Mantendo modelos simples, o software resultante também será simples. O resultado será um software mais fácil de ser integrado, testado e mantido. Além disso, modelos simples são mais fáceis de compreender e criticar, resultando em uma forma contínua de *feedback* que otimiza o resultado final.

Princípio 4. Construa modelos que facilitem alterações. Considere que os modelos mudarão, mas, ao considerar tal fato, não seja relapso. Por exemplo, uma vez que os requisitos serão alterados, há uma tendência a dar pouca atenção a seus modelos. Por quê? Porque se sabe que mudarão de qualquer forma. O problema dessa atitude é que sem um modelo de requisitos razoavelmente completo, criar-se-á um projeto (modelo de projeto) que invariavelmente irá deixar de lado funções e características importantes.

Princípio 5. Seja capaz de estabelecer um propósito claro para cada modelo. Toda vez que criar um modelo, pergunte-se o motivo para tanto. Se você não for capaz de dar justificativas sólidas para a existência do modelo, não desperdice tempo com ele.

Princípio 6. Adapte o modelo desenvolvido ao sistema à disposição. Talvez seja necessário adaptar a notação ou as regras do modelo ao aplicativo; por exemplo, um aplicativo



O objetivo de qualquer modelo é transmitir informações. Para tanto, use um formato consistente. Considere o fato de não estar lá para explicá-lo. Ele deve ser autoexplicativo.

5 Os princípios citados nessa seção foram resumidos e reescritos de acordo com os propósitos do livro.

de videogame pode requerer uma técnica de modelagem diferente daquela utilizada em um software embutido e de tempo real que controla o motor de um automóvel.

Princípio 7. *Crie modelos úteis, mas esqueça a construção de modelos perfeitos.* Ao construir modelos de requisitos e de projetos, um engenheiro de software atinge um ponto de retornos decrescentes. Isto é, o esforço necessário para fazer o modelo absolutamente completo e internamente consistente não vale os benefícios resultantes. Estaria eu sugerindo que a modelagem deve ser descuidada ou de baixa qualidade? A resposta é “não”. Porém, a modelagem deve ser conduzida tendo em vista as etapas de engenharia de software seguintes; iterar indefinidamente para criar um modelo “perfeito” não atende à necessidade de agilidade.

Princípio 8. *Não seja dogmático quanto à sintaxe do modelo. Se esta transmite o conteúdo com sucesso, a representação é secundária.* Embora todos de uma equipe devam tentar usar uma notação consistente durante a modelagem, a característica mais importante do modelo reside em transmitir informações que possibilitem a próxima tarefa de engenharia. Se um modelo viabilizar isso com êxito, a sintaxe incorreta pode ser perdoada.

Princípio 9. *Se os instintos indicam que um modelo não está correto, embora pareça correto no papel, provavelmente há motivos com os quais se preocupar.* Se você for um engenheiro experiente, confie em seus instintos. O trabalho com software nos ensina muitas lições — muitas das quais em um nível subconsciente. Se algo lhe diz que um modelo de projeto parece falho, embora não haja provas explícitas, há motivos para dedicar tempo extra examinando o modelo ou desenvolvendo outro diferente.

Princípio 10. *Obtenha feedback o quanto antes.* Todo modelo deve ser revisado pelos membros da equipe de software. O objetivo das revisões é proporcionar *feedback* que seja usado para corrigir erros de modelagem, alterar interpretações errôneas, e adicionar características ou funções omitidas inadvertidamente.

Princípios de modelagem de requisitos. Ao longo das últimas três décadas, desenvolveu-se um grande número de métodos de modelagem de requisitos. Pesquisadores identificaram problemas de análise de requisitos e suas causas e desenvolveram uma série de notações de modelagem e uma série de “heurísticas” correspondentes para superá-los. Cada um dos métodos de análise possui um ponto de vista particular. Entretanto, todos estão inter-relacionados por uma série de princípios operacionais:

Princípio 1. *O universo de informações de um problema deve ser representado e compreendido.* O universo de informações engloba os dados constantes no sistema (do usuário final, de outros sistemas ou dispositivos externos), os dados que fluem para fora do sistema (via interface do usuário, interfaces de rede, relatórios, gráficos e outros meios) e a armazenagem de dados que coleta e organiza objetos de dados persistentes (isto é, dados que são mantidos permanentemente).

Princípio 2. *As funções que o software desempenha devem ser definidas.* As funções de software oferecem benefício direto aos usuários finais e também suporte interno para fatores visíveis aos usuários. Algumas funções transformam dados que fluem no sistema. Em outros casos, as funções exercem certo nível de controle sobre o processamento interno do software ou sobre elementos de sistema externo. As funções podem ser descritas em diferentes níveis de abstração, desde afirmação geral até uma descrição detalhada dos elementos de processo que devem ser requisitados.

Princípio 3. *O comportamento do software (como consequência de eventos externos) deve ser representado.* O comportamento de um software é comandado por sua interação com o ambiente externo. Alimentações fornecidas pelos usuários finais, informações referentes a controles provenientes de um sistema externo ou dados de monitoramento

PONTO-CHAVE

A modelagem de análise focaliza três atributos de software: informações a serem processadas, função a ser entregue e comportamento a ser apresentado.

“O primeiro problema do engenheiro em qualquer situação de projeto é descobrir qual é realmente o problema.”

Autor desconhecido

coletados de uma rede, todos esses elementos determinam que o software se opere de maneira específica.

Princípio 4. Os modelos que descrevem informações, funções e comportamentos devem ser divididos para que revelem detalhes por camadas (ou hierarquicamente). A *modelagem de requisitos* é a primeira etapa da solução de um problema de engenharia de software. Permite que se entenda melhor o problema e se estabeleçam bases para a solução (projeto). Os problemas complexos são difíceis de resolver em sua totalidade. Por essa razão, deve-se usar a estratégia dividir-e-conquistar. Um problema grande e complexo é dividido em subproblemas até que cada um seja relativamente fácil de ser compreendido. Esse conceito é denominado *fracionamento* ou *separação por interesse* e é uma estratégia-chave na modelagem de requisitos.

Princípio 5. A análise deve partir da informação essencial para o detalhamento da implementação. A *modelagem de requisitos se inicia* pela descrição do problema sob a perspectiva do usuário final. A “essência” do problema é descrita sem levar em consideração como será implementada uma solução. Por exemplo, um jogo de videogame requer que o jogador “instrua” seu protagonista sobre qual direção seguir para continuar conforme se movimenta em situações perigosas. Essa é a essência do problema. O detalhamento da implementação (em geral descrito como parte do modelo de projeto) indica como a essência será implementada. No caso do videogame, talvez fosse usada entrada de voz. Outra alternativa seria utilizar o comando por meio do teclado, ou por joystick ou mouse para uma direção específica, ou um dispositivo com sensor de movimento poderia ser empregado externamente. Aplicando-se tais princípios, um engenheiro de software aborda um problema de forma sistemática. Entretanto, como os princípios são aplicados na prática? Essa pergunta será respondida nos Capítulos 5 a 7.

Princípios da modelagem de projetos. A modelagem de projetos de software é análoga ao planejamento de uma casa feito por um arquiteto. Ela começa pela representação do todo a ser construído (por exemplo, uma representação tridimensional da casa) e, gradualmente, foca os detalhes, oferecendo um roteiro para a sua construção (por exemplo, a estrutura do encanamento). De modo similar, a modelagem de projeto fornece uma variedade de diferentes focos do sistema.

Não há poucos métodos para identificar os vários elementos de um projeto. Alguns são voltados a dados, permitindo que a estrutura de dados determine a arquitetura do programa e dos componentes de processamento resultantes. Outros são voltados para padrões, usando informações a respeito do domínio do problema (da modelagem dos requisitos) para desenvolver os estilos arquitetônicos e os padrões de processamento. Outros, ainda, são voltados a objetos, usando os objetos da área do problema como determinantes para a criação dos métodos e das estruturas de dados que os manipularão. Ainda assim, todos englobam uma série de princípios de projeto que podem ser aplicados independentemente do método empregado:

Princípio 1. O projeto deve ser roteirizado para a modelagem de requisitos.

A modelagem de requisitos descreve a área de informação do problema, funções visíveis ao usuário, desempenho do sistema e um conjunto de classes de requisitos que embala objetos de negócios juntamente com os métodos que ele serve. A modelagem de projeto traduz essa informação em forma de uma arquitetura, um conjunto de subsistemas que implementam funções mais amplas e um conjunto de componentes que são a concretização das classes de requisitos. Os elementos da modelagem de projetos devem ser roteirizados para a modelagem de requisitos.

Princípio 2. Sempre considere a arquitetura do sistema a ser construído.

A arquitetura de software (Capítulo 9) é a espinha dorsal do sistema a ser construído. Afetando interfaces, estruturas de dados, desempenho e fluxo de controle de programas, maneira

“Primeiramente, verifique se o projeto é inteligente e preciso: feito isso, persista resolutamente. Não desista do seu propósito por causa de uma rejeição.

William Shakespeare

WebRef

Comentários mais específicos sobre o processo dos projetos, juntamente com um debate sobre sua estética, poderão ser encontrados em cs.www.edu/~aabyan/Design/.

pela qual os testes podem ser conduzidos, a manutenção do sistema realizada e muito mais. Por todas essas razões, o projeto deve começar com as considerações arquitetônicas. Só depois de a arquitetura ter sido estabelecida devem ser considerados os elementos relativos a componentes.

Princípio 3. O projeto de dados é tão importante quanto o projeto das funções de processamento. O projeto de dados é um elemento essencial do projeto da arquitetura. A forma como os objetos de dados são percebidos no projeto não pode ser deixada ao acaso. Um projeto de dados bem estruturado ajuda a simplificar o fluxo do programa, torna mais fácil a elaboração do projeto e a implementação dos componentes de software, tornando mais eficiente o processamento como um todo.

Princípio 4. As interfaces (tanto internas quanto externas) devem ser projetadas com cuidado. A forma como os dados fluem entre os componentes de um sistema tem muito a ver com a eficiência do processamento, com a propagação de erros e com a simplicidade do projeto. Uma interface bem elaborada facilita a integração e auxilia o responsável pelos testes quanto à validação das funções dos componentes.

Princípio 5. O projeto de interface do usuário deve ser voltado às necessidades do usuário final. Entretanto, em todo caso, deve enfatizar a facilidade de uso. A interface do usuário é a manifestação visível do software. Não importa quão sofisticadas sejam as funções internas, quão amplas sejam as estruturas de dados, quão bem projetada seja a arquitetura; um projeto de interface pobre leva à percepção de que um software é “ruim”.

Princípio 6. O projeto no nível de componentes deve ser funcionalmente independente. Independência funcional é uma medida para a “mentalidade simplificada” de um componente de software. A funcionalidade entregue por um componente deve ser coesa — isto é, focarem uma, e somente uma, função ou subfunção.⁶

Princípio 7. Os componentes devem ser relacionados livremente tanto entre componentes quanto com o ambiente externo. O relacionamento é obtido de várias maneiras — via interface de componentes, por meio de mensagens, através de dados em geral. Na medida em que o nível de correlacionamento aumenta, a tendência para a propagação do erro também aumenta e a manutenção geral do software decresce. Portanto, o relacionamento entre componentes deve ser mantido tão baixo quanto possível.

Princípio 8. Representações de projetos (modelos) devem ser de fácil compreensão. A finalidade de projetos é transmitir informações aos desenvolvedores que farão a codificação, àqueles que irão testar o software e para outros que possam vir a dar manutenção futuramente. Se o projeto for de difícil compreensão, não servirá como meio de comunicação efetivo.

Princípio 9. O projeto deve ser desenvolvido iterativamente. A cada iteração, o projetista deve se esforçar para obter maior grau de simplicidade. Como todas as atividades criativas, a elaboração de um projeto ocorre de forma iterativa. As primeiras iterações são realizadas para refinar o projeto e corrigir erros, entretanto, as iterações finais devem dirigir esforços para tornar o projeto tão simples quanto possível.

Quando tais princípios são aplicados apropriadamente, elabora-se um projeto com fatores de qualidade tanto externos quanto internos [Mye78]. *Fatores externos de qualidade* são as propriedades que podem ser prontamente notadas pelos usuários (por exemplo, velocidade, confiabilidade, correção, usabilidade). Os *fatores internos de qualidade* são de extrema importância para os engenheiros de software. Conduzem a um projeto de alta qualidade do ponto de vista técnico. Para obter fatores de qualidade internos, o projetista deve entender sobre conceitos básicos de projeto (Capítulo 8).

“As diferenças não são insignificantes — são bem semelhantes às diferenças entre Salieri e Mozart. Estudos após estudos demonstram que os melhores projetistas produzem estruturas mais rápidas, menores, mais simples, mais claras e com menos esforço.”

Frederick P.
Brooks

6 Mais informações a respeito de coesão podem ser encontradas no Capítulo 8.

“Por muito tempo em minha vida, fui um observador em relação ao mundo do software, espionando furtivamente os códigos poluídos de outras pessoas. Ocasionalmente, encontro uma joia verdadeira, um programa bem estruturado, escrito em um estilo consistente, livre de gambiarras, desenvolvido de modo que cada componente seja simples, organizado e projetado de forma que o produto possa ser facilmente alterado.”

David Parnas



Evite desenvolver um programa elegante que resolva o problema errado. Preste particular atenção ao primeiro princípio referente à preparação.

WebRef

Uma ampla variedade de links para padrões de codificação pode ser encontrada no site www.literateprogramming.com/fpstyle.html.

4.3.4 Princípios de construção

A atividade de construção engloba um conjunto de tarefas de codificação e testes que conduzem ao software operacional pronto para ser entregue ao cliente e ao usuário final. Na atividade moderna de engenharia de software, a codificação pode ser: (1) a criação direta do código-fonte da linguagem de programação (por exemplo, Java), (2) a geração automática de código-fonte usando uma representação intermediária semelhante a um projeto do componente a ser construído ou então (3) a geração automática de código executável usando uma “linguagem de programação de quarta geração” (por exemplo, Visual C++).

O foco inicial dos testes é voltado para componentes, com frequência denominado *teste de unidade*. Outros níveis de testes incluem: (1) *teste de integração* (realizado à medida que o sistema é construído), (2) *teste de validação* que avalia se os requisitos foram atendidos para o sistema completo (ou incremento de software) e (3) *teste de aceitação conduzido* pelo cliente voltado para empregar todos os fatores e funções requisitados. A série de princípios e conceitos fundamentais a seguir é aplicável à codificação e aos testes:

Princípios de codificação. Os princípios que regem a codificação são intimamente alinhados com o estilo de programação, com as linguagens e com os métodos de programação. Entretanto, há uma série de princípios fundamentais que podem ser estabelecidos:

Princípios de preparação: *Antes de escrever uma única linha de código, certifique-se de que*

- Compreendeu bem o problema a ser solucionado.
- Compreendeu bem os princípios e conceitos básicos sobre o projeto.
- Escolheu uma linguagem de programação adequada às necessidades do software a ser desenvolvido e ao ambiente em que ele irá operar.
- Selecionou um ambiente de programação que forneça ferramentas que tornarão seu trabalho mais fácil.
- Elaborou um conjunto de testes de unidades que serão aplicados assim que o componente codificado estiver completo.

Princípios de programação: *Ao começar a escrever código*

- Restrinja seus algoritmos seguindo a prática de programação estruturada [Boh00].
- Considere o emprego de uso de programação pareada.
- Selecione estruturas de dados que venham ao encontro das do projeto.
- Domine a arquitetura de software e crie interfaces consistentes com ela.
- Mantenha a lógica condicional tão simples quanto possível.
- Crie “loops” agrupados de tal forma que testes sejam facilmente aplicáveis.
- Escolha denominações de variáveis significativas e obedeça a outros padrões de codificação locais.
- Faça uma documentação que seja autodocumentável.
- Crie uma disposição (layout) visual (por exemplo, recorte e linhas em branco) que auxilie a compreensão.

Princípios de validação: *Após completar a primeira etapa de codificação, certifique-se de*

- Aplicar uma revisão de código quando for apropriado.
- Realizar testes de unidades e corrigir erros ainda não identificados.
- Refazer a codificação.

Mais livros foram escritos sobre programação (codificação) e sobre os princípios e os conceitos que a guiam do que qualquer outro tópico do processo de software. Livros sobre o tema incluem trabalhos antigos em estilo de programação [Ker78], construção prática de software [McC04],

pérolas da programação [Ben99], a arte de programar [Knu98], elementos da programação pragmática [Hun99] e muitos, muitos outros assuntos. Um debate amplo sobre esses princípios e conceitos foge do escopo deste livro. Caso haja maior interesse, examine uma ou mais das referências indicadas.

Princípios de testes. Em um clássico sobre testes de software, Glen Myers [Mye79] estabelece uma série de regras que podem servir, bem como objetivos da atividade de testes:

 **Quais são os objetivos dos testes de software?**

- Teste consiste em um processo de executar um programa com o intuito de encontrar um erro.
- Um bom pacote de testes é aquele em que há uma alta probabilidade de encontrar um erro ainda não descoberto.
- Um teste bem-sucedido é aquele que revela um novo erro.



Em um contexto mais amplo de projeto de software, atente para iniciar “no geral”, focalizando a arquitetura de software, e terminar “no particular”, focalizando os componentes. Para testes, simplesmente reverta o foco e teste seu desenvolvimento.

Esses objetivos implicam uma mudança radical de ponto de vista para alguns desenvolvedores. Vão contra a visão comumente difundida de que um teste bem-sucedido é aquele em que nenhum erro é encontrado. Seu objetivo é o de projetar testes que descubram, sistematicamente, diferentes classes de erros, consumindo o mínimo de esforço e tempo.

Se testes forem conduzidos com êxito (de acordo com os objetivos declarados previamente), irão encontrar erros no software. Como benefício secundário, os testes demonstram que as funções do software estão funcionando de acordo com as especificações e que os requisitos relativos ao desempenho e ao comportamento parecem estar sendo atingidos. Os dados coletados durante os testes fornecem um bom indício da confiabilidade do software, assim como fornecem a indicação da qualidade do software como um todo. Entretanto, os testes não são capazes de mostrar a ausência de erros e defeitos; podendo apenas mostrar que erros e defeitos de software estão presentes. É importante manter isso em mente (do que negligenciá-lo) enquanto os testes estão sendo aplicados.

Davis [Dav95b] sugere um conjunto de princípios de testes⁷ adaptados para este livro:

Princípio 1. Todos os testes devem estar alinhados com os requisitos do cliente.⁸

O objetivo dos testes é desvendar erros. Consta-se que os efeitos mais críticos do ponto de vista do cliente são aqueles que conduzem a falhas no programa quanto a seus requisitos.

Princípio 2. Os testes devem ser planejados muito antes de ser iniciados. O planejamento dos testes (Capítulo 17) pode começar assim que o modelo de requisitos estiver completo. A definição detalhada dos pacotes de teste pode começar tão logo o modelo de projeto tenha sido solidificado. Portanto, todos os testes podem ser planejados e projetados antes que qualquer codificação tenha sido gerada.

Princípio 3. O princípio de Pareto se aplica a testes de software. Neste contexto o princípio de Pareto implica que 80% de todos os erros revelados durante testes provavelmente estarão alinhados a aproximadamente 20% de todos os componentes de programa. O problema, evidentemente, consiste em isolar os componentes suspeitos e testá-los por completo.

Princípio 4. Os testes devem começar “em particular” e progredir rumo ao teste “em geral”. Os primeiros testes planejados e executados geralmente focam os componentes individuais. À medida que progridem, o foco muda para tentar encontrar erros em grupos de componentes integrados e, posteriormente, no sistema inteiro.

Princípio 5. Testes exaustivos são impossíveis. A quantidade de trocas de direções, mesmo para um programa de tamanho moderado, é excepcionalmente grande. Por essa

⁷ Apenas um pequeno subconjunto dos princípios de testes de David é citado aqui. Para maiores informações, veja [Dav95b].

⁸ Esse princípio refere-se a testes funcionais, por exemplo, testes que se concentram em requisitos. Os testes estruturais (testes que se concentram nos detalhes lógicos ou arquitetônicos) não podem referir-se a requisitos específicos diretamente.

razão, é impossível executar todas as rotas durante os testes. Sendo possível, no entanto, cobrir adequadamente a lógica do programa e garantir que todas as condições referentes ao projeto no nível de componentes sejam exercidas.

4.3.5 Princípios de disponibilização

Como observado anteriormente, na Parte 1 deste livro, a disponibilização envolve três ações: entrega, suporte e feedback. Pelo fato de os modernos modelos de processos de software serem, em sua natureza, evolucionários ou incrementais, a disponibilização não ocorre imediatamente, mas sim, em muitas vezes, quando o software segue para sua finalização. Cada ciclo de entrega propicia ao cliente e ao usuário um incremento de software operacional que fornece fatores e funções utilizáveis. Cada ciclo de suporte fornece assistência humana e documentação para todas as funções e fatores introduzidos durante todos os ciclos de disponibilização até o presente. Cada ciclo de feedback fornece à equipe de software importante roteiro que resulta em alteração de funções, elementos e abordagem adotados para o próximo incremento.

A entrega para um incremento de software representa um marco importante para qualquer projeto de software. Uma série de princípios essenciais deve ser seguida enquanto a equipe se prepara para a entrega de um incremento:



Certifique-se de que seu cliente saiba o que esperar antes da entrega de um incremento de software. Caso contrário, com certeza ele contará mais do que você poderá entregar.

Princípio 1. As expectativas dos clientes para o software devem ser gerenciadas.

Muitas vezes, o cliente espera mais do que a equipe havia prometido entregar e, imediatamente, ocorre o desapontamento. Isso resulta em feedback não produtivo e arruína o moral da equipe. Em seu livro sobre gerenciamento de expectativas, Naomi Karten [Kar94] afirma: “O ponto de partida para administrar expectativas consiste em se tornar mais consciencioso sobre como e o que vai comunicar.” Ela sugere que um engenheiro de software deve ser cauteloso em relação ao envio de mensagens conflituosas ao cliente (por exemplo, prometer mais do que pode entregar racionalmente no prazo estabelecido ou entregar mais do que o prometido para determinado incremento e, em seguida, menos do que prometera para o próximo).

Princípio 2. Um pacote de entrega completo deve ser auditorado e testado. Um CD-ROM ou outra mídia (inclusive downloads de Web) contendo todo o software executável, arquivos de dados de suporte, documentos de suporte e outras informações relevantes devem ser completamente checados e testados por meio de uma versão beta com os reais usuários. Todos os roteiros de instalação e outros itens operacionais devem ser rodados inteiramente em tantas configurações computacionais quanto forem possíveis (isto é, hardware, sistemas operacionais, dispositivos periféricos, disposições de rede).

Princípio 3. Deve-se estabelecer uma estrutura de suporte antes da entrega do software. Um usuário final conta com recebimento de informações acuradas e com responsabilidade caso surja um problema. Se o suporte for local, ou, pior ainda, inexistente, imediatamente o cliente ficará insatisfeito. O suporte deve ser planejado, e seus materiais devem estar preparados, e mecanismos para manutenção de registros apropriados devem estar determinados para que a equipe de software possa oferecer uma avaliação de qualidade das formas de suporte solicitadas.

Princípio 4. Material adequado referente a instruções deve ser fornecido aos usuários finais. A equipe de software deve entregar mais do que o software em si. Auxílio em treinamento de forma adequada (se solicitado) deve ser desenvolvido; orientações quanto a problemas inesperados devem ser oferecidas; quando necessário, é importante editar uma descrição acerca de “diferenças existentes no incremento de software”.⁹

Princípio 5. Software com bugs (erros), primeiro, deve ser corrigido, e, depois, entregue. Sob a pressão relativa a prazo, muitas empresas de software entregam incrementos

9 Durante a atividade de comunicação, a equipe de software deve determinar quais materiais de apoio os usuários desejam.

de baixa qualidade, notificando o cliente que os bugs “serão corrigidos na próxima versão”. Isso é um erro. Há um ditado no mercado de software: “Os clientes esquecerão a entrega de um produto de alta qualidade em poucos dias, mas jamais esquecerão os problemas causados por um produto de baixa qualidade. O software os faz lembrar disso todos os dias”.

O software entregue propicia benefício ao usuário final, mas este também fornece feedback proveitoso para a equipe de software. À medida que um incremento é colocado em uso, os usuários finais devem ser encorajados a tecer comentários acerca das características e funções, facilidade de uso, confiabilidade e quaisquer outras características apropriadas.

4.4 RESUMO

A prática de engenharia de software envolve princípios, conceitos, métodos e ferramentas aplicados por engenheiros da área ao longo de todo o processo de desenvolvimento. Cada projeto de engenharia de software é diferente. Ainda assim, uma gama de princípios genéricos se aplica ao processo como um todo e à prática de cada atividade metodológica independentemente do projeto ou do produto.

Um conjunto de princípios essenciais auxilia na aplicação de um processo de software significativo e na execução de métodos efetivos de engenharia de software. Quanto ao processo, os princípios essenciais estabelecem uma base filosófica que orienta a equipe durante essa fase de desenvolvimento. Quanto ao nível relativo à prática, os princípios estabelecem uma série de valores e regras que servem como guia ao se analisar um problema, projetar uma solução, implementar e testar uma solução e, por fim, disponibilizar o software para a sua comunidade de usuários.

Princípios de comunicação enfatizam a necessidade de reduzir ruído e aumentar a dimensão conforme o diálogo entre o desenvolvedor e o cliente progride. Ambas as partes devem colaborar para que ocorra a melhor comunicação.

Princípios de planejamento proporcionam roteiros para a construção do melhor mapa para a jornada rumo a um sistema ou produto completo. O plano pode ser projetado para um único incremento de software ou pode ser definido para o projeto inteiro. Independentemente da situação, deve indicar o que será feito, quem o fará e quando o trabalho estará completo.

Modelagem abrange tanto análise quanto projeto, descrevendo representações do software que se tornam progressivamente mais detalhadas. O objetivo dos modelos é solidificar a compreensão do trabalho a ser feito e providenciar orientação técnica aos implementadores do software. Os princípios de modelagem servem como infraestrutura para os métodos e para a notação utilizada para criar representações do software.

Construção incorpora um ciclo de codificação e testes no qual o código-fonte para um componente é gerado e testado. Os princípios de codificação definem ações genéricas que devem ocorrer antes da codificação ser feita, enquanto está sendo criada e após estar completa. Embora haja muitos princípios de testes, apenas um é dominante: teste consiste em um processo de execução de um programa com o intuito de encontrar um erro.

Disponibilização ocorre na medida em que cada incremento de software é apresentado ao cliente e engloba a entrega, o suporte e o feedback. Os princípios fundamentais para a entrega consideram o gerenciamento das expectativas dos clientes e o fornecimento ao cliente de informações de suporte apropriadas sobre o software. O suporte exige preparação antecipada. Feedback permite ao cliente sugerir mudanças que tenham valor agregado e fornecer ao desenvolvedor informações para o próximo ciclo de engenharia de software.

PROBLEMAS E PONTOS A PONDERAR

4.1. Uma vez que o foco em qualidade demanda recursos e tempo, é possível ser ágil e ainda assim manter o foco em qualidade?

- 4.2. Dos oito princípios básicos que orientam um processo (discutido na Seção 4.2.1), qual você acredita ser o mais importante?
- 4.3. Descreva o conceito de *separação por interesses* com suas próprias palavras.
- 4.4. Um importante princípio de comunicação afirma “prepare-se antes de se comunicar”. Como essa preparação se manifesta no trabalho prévio que você realiza? Quais produtos de trabalho podem resultar da preparação antecipada?
- 4.5. Pesquise sobre “facilitação” para a atividade de comunicação (use as referências fornecidas ou outras) e prepare um conjunto de passos focados somente em facilitação.
- 4.6. Em que a comunicação ágil difere da tradicional em engenharia de software? Em que ela é similar?
- 4.7. Por que é necessário “seguir em frente”?
- 4.8. Pesquise sobre “negociação” para a atividade de comunicação e prepare uma série de etapas concentrando-se apenas na negociação.
- 4.9. Descreva o que significa “particularidade” no contexto do cronograma de um projeto.
- 4.10. Por que os modelos são importantes no trabalho de engenharia de software? São eles sempre necessários? Existem qualificadores para sua resposta sobre necessidade?
- 4.11. Quais são os três “domínios” considerados durante a modelagem de requisitos?
- 4.12. Tente acrescentar mais um princípio àqueles determinados para a codificação na Seção 4.3.4.
- 4.13. Em que consiste um teste bem-sucedido?
- 4.14. Você concorda ou discorda com a seguinte afirmação: “Uma vez que vários incrementos são entregues ao cliente, por que se preocupar com a qualidade nos incrementos iniciais — pode-se corrigir os problemas em iterações posteriores”. Justifique sua resposta.
- 4.15. Por que o feedback é importante para uma equipe de software?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A comunicação com o cliente é uma atividade crítica na engenharia de software, embora poucos profissionais invistam em sua leitura. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) apresenta uma gama de padrões úteis que tratam de problemas de comunicação. Sutliff (*User-Centred Requirements Engineering*, Springer, 2002) se concentra muito nos desafios relacionados com a comunicação. Livros como os de Weigers (*Software Requirements*, 2. ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) e Karten [Kar94] dão uma excelente visão sobre métodos para interação efetiva com os clientes. Embora a obra não se concentre em software, Hooks e Farry (*Customer Centered Products*, American Management Association, 2000) apresentam úteis diretrizes genéricas para a comunicação com os clientes. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) enfatiza uma “equipe conjunta” entre clientes e desenvolvedores que desenvolvem requisitos de forma colaborativa. Sommerville e Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) discutem técnicas e conceitos de “suscitação”, bem como outros princípios de engenharia de requisitos.

Conceitos e princípios de comunicação e planejamento são considerados em vários livros de gerenciamento de projetos. Entre eles podemos citar: Bechtold (*Essentials of Software Project Management*, 2. ed., Management Concepts, 2007), Wisoeki (*Effective Project Management: Traditional, Adaptive, Extreme*, 4. ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principles for Success*, BookSurge Publishing, 2006), Hughes (*Software Project Management*, McGraw-Hill, 2005) e Stellman e Greene (*Applied Software Project Management*, O'Reilly Media, Inc., 2005).

Davis [Dav95] compilou um excelente conjunto de princípios de engenharia de software. Além disso, praticamente todo livro sobre engenharia de software contém uma discussão

proveitosa sobre conceitos e princípios para análise, projeto e testes. Entre os livros mais largamente usados (além deste livro, é claro!), temos:

- Abran, A. e J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.
- Christensen, M. e R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.
- Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.
- Pfleeger, S., *Software Engineering: Theory and Practice*, 3. ed., Prentice-Hall, 2005.
- Schach, S., *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*, McGraw-Hill, 7. ed., 2008.
- Sommerville, I., *Software Engineering*, 8. ed., Addison-Wesley, 2006.

Esses livros também apresentam uma discussão detalhada sobre princípios de modelagem e construção.

Princípios de modelagem são considerados em muitos textos dedicados à análise de requisitos e/ou projeto de software. Livros como os de Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice*, Wiley, 2004), Penker e Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) discutem os princípios e métodos de modelagem.

A obra de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990) é uma leitura obrigatória para todo engenheiro de software que pretenda realizar trabalhos de projeto. Winograd e seus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaram um excelente conjunto de artigos que tratam das questões práticas no projeto de software. Constantine e Lockwood (*Software for Use*, Addison-Wesley, 1999) apresentam os conceitos associados ao “projeto centrado no usuário”. Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) traz uma discussão filosófica proveitosa sobre a natureza do projeto e o impacto das decisões na qualidade e na habilidade de a equipe produzir software que forneça grande valor a seu cliente. Stahl e seus colegas (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) discutem os princípios do desenvolvimento dirigido por modelos.

Centenas de livros tratam um ou mais elementos da atividade de construção. Kernighan e Plauger [Ker78] escreveram um clássico sobre estilo de programação, McConnell [McC93] apresenta diretrizes pragmáticas para a construção prática de software, Bentley [Ben99] sugere uma ampla gama de pérolas da programação, Knuth [Knu99] escreveu uma série clássica de três volumes sobre a arte de programar e Hunt [Hun99] sugere diretrizes de programação pragmáticas.

Myers e seus colegas (*The Art of Software Testing*, 2. ed., Wiley, 2004) fizeram uma revisão importante de seu texto clássico e discutem vários princípios de testes importantes. Livros como os de Perry (*Effective Methods for Software Testing*, 3. ed., Wiley, 2006), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner e seus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) e Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) introduzem, cada um deles, importantes conceitos e princípios para testes e muita orientação pragmática.

Uma ampla gama de fontes de informação sobre a prática da engenharia de software se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevantes à prática da engenharia de software, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

5

ENGENHARIA DE REQUISITOS

CONCEITOS-CHAVE

casos de uso.....	137
colaboração.....	132
concepção.....	127
disponibilização da função de qualidade.....	136
elaboração.....	128
engenharia de requisitos.....	127
especificação.....	129
gestão dos requisitos.....	130
interessados.....	131
levantamento.....	128
levantamento dos requisitos.....	133
modelo de análise.....	142
negociação.....	128
padrões de análise.....	145
pontos de vista.....	131
produtos resultantes.....	137
validação.....	129
validação dos requisitos.....	146

Entender os requisitos de um problema está entre as tarefas mais difíceis enfrentadas por um engenheiro de software. Ao pensar nisso pela primeira vez, a engenharia de requisitos não parece assim tão difícil. Afinal de contas, o cliente não sabe o que é necessário? Os usuários finais não deveriam ter um bom entendimento das características e funções que trarão benefícios? Surpreendentemente, em muitos casos a resposta a essas perguntas é “não”. E mesmo se os clientes e usuários finais fossem explícitos quanto às suas necessidades, essas mudariam ao longo do projeto.

No prefácio de um livro de Ralph Young [You01] sobre práticas de necessidades eficazes, escrevi:

É o seu pior pesadelo. Um cliente entra no seu escritório, se senta, te olha diretamente nos olhos e diz: “Eu sei que você imagina que entendeu aquilo que eu lhe disse, mas o que você não compreende é que aquilo que eu lhe disse não era exatamente aquilo que eu quis dizer”. Invariavelmente, isso acontece no final do projeto, após compromissos de prazos de entrega terem sido estabelecidos, reputações estarem na mira e muito dinheiro estar em jogo.

Todos que já trabalharam na área de software e sistemas há alguns anos viveram esse pesadelo e, mesmo assim, poucos aprenderam a livrar-se dele. Passamos por muitas dificuldades ao tentar extrair os requisitos de nossos clientes. Temos dificuldades para entender as informações obtidas. Normalmente registramos os requisitos de uma forma desorganizada e investimos muito pouco tempo verificando aquilo que registramos. Deixamos que as mudanças nos controlem, em vez de estabelecermos mecanismos para controlar as mudanças. Em suma, falhamos ao estabelecer uma base sólida para o sistema ou software. Cada um desses problemas é desafiador. Quando combinados, o panorama é assustador até mesmo para os gerentes e profissionais mais experientes. Mas soluções de fato existem.

PANORAMA

O que é? Antes de iniciar qualquer trabalho técnico, é uma boa ideia aplicar um conjunto de tarefas de engenharia de requisitos. Estas levam a um entendimento de qual será o impacto do software sobre o negócio, o que o cliente quer e como os usuários finais irão interagir com o software.

Quem realiza? Engenheiros de software (algumas vezes conhecidos no mundo da TI como engenheiros de sistemas ou “analistas”) e outros interessados no projeto (gerentes, clientes, usuários finais), todos participam da engenharia de requisitos.

Por que é importante? Projetar e construir um programa de computador elegante que resolva o problema errado não atende às necessidades de ninguém. É por isso que é importante entender o que o cliente quer antes de começar a projetar e construir um sistema baseado em computador.

Quais são as etapas envolvidas? A engenharia de requisitos começa com a concepção — uma tarefa que define o escopo e a natureza do problema a ser resolvido. Ela prossegue para o levantamento — uma tarefa que ajuda os interessados a definir o que é necessário e, então, para a elabo-

ração — onde os requisitos básicos são refinados e modificados. À medida que os interessados definem o problema, ocorre a negociação — quais são as prioridades, o que é essencial, quando é necessário? Por fim, o problema é especificado de algum modo e, então, revisado ou validado para garantir que seu entendimento e o entendimento dos interessados sobre o problema coincidam.

Qual é o artefato? O objetivo da engenharia de requisitos é fornecer a todas as partes um entendimento escrito do problema. Isso pode ser alcançado por meio de uma série de artefatos: cenários de uso, listas de funções e características, modelos de análise ou uma especificação.

Como garantir que o trabalho foi realizado corretamente? Os artefatos da engenharia de requisitos são revisados com os interessados para garantir que aquilo que você entendeu é realmente aquilo que eles queriam dizer. Um alerta: mesmo depois de todas as partes terem entrado em acordo, as coisas vão mudar e continuarão mudando ao longo de todo o projeto.

É razoável argumentar que as técnicas que discutiremos neste capítulo não são uma verdadeira “solução” para os desafios que acabamos de citar. Mas elas fornecem efetivamente uma abordagem consistente para lidar com tais desafios.

5.1 ENGENHARIA DE REQUISITOS

“A parte mais difícil ao construir um sistema de software é decidir o que construir. Nenhuma parte do trabalho afeta tanto o sistema resultante se for feita a coisa errada. Nenhuma outra parte é mais difícil de consertar depois.”

Fred Brooks

PONTO-CHAVE

A engenharia de requisitos estabelece uma base sólida para o projeto e para a construção. Sem ela, o software resultante tem grande probabilidade de não atender às necessidades do cliente.



AVISO
Espere ter de realizar um pouco de projeto durante o trabalho de levantamento de requisitos e um pouco de trabalho de levantamento de requisitos durante o projeto.

Projetar e construir software é desafiador, criativo e pura diversão. Na realidade, construir software é tão cativante que muitos desenvolvedores desejam iniciar logo, antes de terem um claro entendimento daquilo que é necessário. Eles argumentam que as coisas ficarão mais claras à medida que forem construindo o software, que os interessados no projeto serão capazes de entender a necessidade apenas depois de examinar as primeiras iterações do software, que as coisas mudam tão rápido que qualquer tentativa de entender os requisitos de forma detalhada será uma perda de tempo, que o primordial é produzir um programa que funcione e que todo o resto é secundário. O que torna esses argumentos tentadores é que contêm elementos de verdade.¹ Porém, cada um apresenta pontos fracos e pode levar um projeto ao fracasso.

O amplo espectro de tarefas e técnicas que levam a um entendimento dos requisitos é denominado *engenharia de requisitos*. Na perspectiva do processo de software, a engenharia de requisitos é uma ação de engenharia de software importante que se inicia durante a atividade de comunicação e continua na de modelagem. Ela deve ser adaptada às necessidades do processo, do projeto, do produto e das pessoas que estão realizando o trabalho.

A engenharia de requisitos constrói uma ponte para o projeto e para a construção. Mas onde começa essa ponte? Alguém pode argumentar que ela começa na base dos interessados no projeto (por exemplo, gerentes, clientes, usuários finais), em que é definida a necessidade do negócio, são descritos cenários de usuários, delineadas funções e recursos e identificadas restrições de projeto. Outros poderiam sugerir que ela se inicia com uma definição mais abrangente do sistema, em que o software é apenas um componente do domínio de sistema mais abrangente. Porém, independentemente do ponto de partida, a jornada através da ponte nos leva bem à frente no projeto, permitindo que examinemos o contexto do trabalho de software a ser realizado; as necessidades específicas que o projeto e a construção devem atender; as prioridades que orientam a ordem na qual o trabalho deve ser completado e as informações, funções e comportamentos que terão um impacto profundo no projeto resultante.

A engenharia de requisitos fornece o mecanismo apropriado para entender aquilo que o cliente deseja, analisando as necessidades, avaliando a viabilidade, negociando uma solução razoável, especificando a solução sem ambiguidades, validando a especificação e gerenciando as necessidades à medida que são transformadas em um sistema operacional [Tha97]. Ela abrange sete tarefas distintas: concepção, levantamento, elaboração, negociação, especificação, validação e gestão. É importante notar que algumas delas ocorrem em paralelo e todas são adaptadas às necessidades do projeto.

Concepção. Como um projeto de software é iniciado? Existe algum evento único que se torna o catalisador para um novo produto ou sistema de computador ou a necessidade evolui ao longo do tempo? Não há nenhuma resposta definitiva para tais perguntas. Em alguns casos, uma conversa informal é tudo o que é preciso para precipitar um trabalho de engenharia de software. Porém, em geral, a maioria dos projetos começa quando é identificada a necessidade do negócio ou é descoberto um novo serviço ou mercado potencial. Interessados da comunidade de negócios (por exemplo, gerentes comerciais, pessoal de marketing, gerentes de produto) definem um plano de negócio para a ideia, tentam identificar o tamanho do mercado, fazem uma análise de

¹ Isto é particularmente verdadeiro para projetos pequenos (menos de um mês) e trabalhos de software menores e relativamente simples. À medida que o software cresce em tamanho e complexidade, tais argumentos caem por terra.

“As sementes das principais catástrofes de software são normalmente semeadas nos três primeiros meses do projeto de software.”

Caper Jones

viabilidade aproximada e identificam uma descrição operacional do escopo do projeto. Todas as informações estão sujeitas a mudanças, porém é suficiente para suscitar discussões com a organização de engenharia de software.²

Na concepção³ do projeto, estabelecemos um entendimento básico do problema, as pessoas que querem uma solução, a natureza da solução desejada e a eficácia da comunicação e colaboração preliminares entre os demais interessados e a equipe de software.

Levantamento. Certamente parece bastante simples — pergunte ao cliente, aos usuários e aos demais interessados quais são os objetivos para o sistema ou produto, o que deve ser alcançado, como o sistema ou produto atende às necessidades da empresa e, por fim, como o sistema ou produto deve ser utilizado no dia a dia. Mas isso não é simples — na verdade, é muito difícil.

Christel e Kang [Cri92] identificaram uma série de problemas que são encontrados durante o levantamento:

- **Problemas de escopo.** Os limites do sistema são definidos de forma precária ou os clientes/usuários especificam detalhes técnicos desnecessários que podem confundir, em vez de esclarecer, os objetivos globais do sistema.
- **Problemas de entendimento.** Os clientes/usuários não estão completamente certos do que é preciso, têm um entendimento inadequado das capacidades e limitações de seus ambientes computacionais, não possuem um entendimento completo do domínio do problema, têm problemas para transmitir suas necessidades ao engenheiro de sistemas, omitem informações que acreditam ser “óbvias”, especificam requisitos que conflitam com as necessidades de outros clientes/usuários ou especificam requisitos que são ambíguos ou impossíveis de ser testados.
- **Problemas de volatilidade.** Os requisitos mudam com o tempo. Para ajudar a superar esses problemas, devemos abordar o levantamento de requisitos de forma organizada.

 **Por que é difícil conseguir um claro entendimento daquilo que o cliente quer?**



A elaboração é uma coisa boa, porém é preciso saber quando parar. O segredo é descrever o problema de maneira que estabeleça uma base sólida para o projeto. Caso passe desse ponto, você estará realizando um projeto.

Elaboração. As informações obtidas do cliente durante as fases de concepção e levantamento são expandidas e refinadas durante a elaboração. Essa tarefa concentra-se no desenvolvimento de um modelo de requisitos refinado (Capítulos 6 e 7) que identifique os diversos aspectos da função, do comportamento e das informações do software.

A elaboração é guiada pela criação e refinamento de cenários de usuários que descrevem como o usuário final (e outros atores) irão interagir com o sistema. Cada cenário de usuário é analisado sintaticamente para extrair classes de análise — entidades do domínio de negócio visíveis para o usuário final. Os atributos de cada classe de análise são definidos, e os serviços⁴ exigidos por cada classe são identificados. As relações e a colaboração entre as classes são identificadas e uma variedade de diagramas suplementares é produzida.

Negociação. Não é incomum clientes e usuários pedirem mais do que pode ser alcançado, dados os recursos limitados do negócio. Também é relativamente comum diferentes clientes ou usuários proporem necessidades conflitantes, argumentando que sua versão é “essencial para nossas necessidades especiais”.

É preciso conciliar esses conflitos por meio de um processo de negociação. Devemos solicitar aos clientes, usuários e outros interessados para que ordenem seus requisitos e discutam em termos de prioridade. Usando uma abordagem iterativa que priorize os requisitos, avalie seus



Em uma negociação efetiva não existem ganhadores, nem perdedores. Ambas as partes ganham, pois é solidificado um “trato” que ambas as partes aceitam.

² Se um sistema baseado em computador deve ser desenvolvido, as discussões começam no contexto de um processo de engenharia de sistemas. Para uma discussão detalhada da engenharia de sistemas, visite o site que acompanha este livro.
³ Lembre-se de que o Processo Unificado (Capítulo 2) define uma “fase de concepção” mais ampla que engloba as tarefas de concepção, levantamento e elaboração discutidas neste capítulo.
⁴ Um serviço manipula os dados encapsulados pela classe. Os termos *operação* e *método* também são usados. Caso não esteja familiarizado com os conceitos da orientação a objetos, é apresentada uma introdução básica no Apêndice 2.

custos e riscos, bem como trate dos conflitos internos, requisitos são eliminados, combinados e/ou modificados, de modo que cada parte atinja certo nível de satisfação.

Especificação. No contexto de sistemas (e software) baseados em computadores, o termo *especificação* assume diferentes significados para pessoas diferentes. Especificação pode ser um documento por escrito, um conjunto de modelos gráficos, um modelo matemático formal, um conjunto de cenários de uso, um protótipo ou qualquer combinação dos fatores citados.

Alguns sugerem que “modelo-padrão” [Som97] deve ser desenvolvido e utilizado para a especificação, argumentando que isso leva a requisitos que são apresentados de forma consistente e, portanto, mais compreensível. Entretanto, algumas vezes é necessário permanecer flexível quando uma especificação deve ser desenvolvida. Para sistemas grandes, um documento escrito, combinando descrições em linguagem natural e modelos gráficos, pode ser a melhor abordagem. Entretanto, talvez sejam necessários apenas cenários de uso para produtos ou sistemas menores que residem em ambientes técnicos bem entendidos.

PONTO-CHAVE

A formalidade e o formato de uma especificação variam com o tamanho e a complexidade do software a ser construído.



Modelo de especificação de requisitos de software

Uma especificação de requisitos de software (*software requirements specification*, SRS) é um documento criado quando uma descrição detalhada de todos os aspectos do software a ser construído deve ser especificada antes de o projeto começar. É importante notar que uma SRS formal nem sempre é por escrito. Na realidade, há várias ocasiões em que o esforço gasto em uma SRS talvez fosse mais bem aproveitado em outras atividades de engenharia de software. Entretanto, quando um software for desenvolvido por terceiros, quando uma falta de especificação criar graves problemas de negócio, ou quando um sistema for extremamente complexo ou crítico para o negócio, será justificável uma SRS.

Karl Wiegers [Wie03] da Process Impact Inc. desenvolveu uma planilha bastante útil (disponível em www.processimpact.com/process_assets/srs_template.doc), que pode servir como diretriz para aqueles que precisam criar uma SRS completa. Uma descrição geral por tópicos é apresentada a seguir:

Sumário

Histórico de revisão

1. Introdução

- 1.1 Propósito
- 1.2 Convenções do documento
- 1.3 Público-alvo e sugestões de leitura
- 1.4 Escopo do projeto
- 1.5 Referências

2. Descrição geral

- 2.1 Perspectiva do produto
- 2.2 Características do Produto

- 2.3 Classes de usuários e características
- 2.4 Ambiente operacional
- 2.5 Restrições de projeto e implementação
- 2.6 Documentação para usuários
- 2.7 Hipóteses e dependências

3. Características do sistema

- 3.1 Características do sistema 1
- 3.2 Características do sistema 2 (e assim por diante)

4. Requisitos de interfaces externas

- 4.1 Interfaces do usuário
- 4.2 Interfaces de hardware
- 4.3 Interfaces de software
- 4.4 Interfaces de comunicação

5. Outros requisitos não funcionais

- 5.1 Necessidades de desempenho
- 5.2 Necessidades de proteção
- 5.3 Necessidades de segurança
- 5.4 Atributos de qualidade de software

6. Outros requisitos

Apêndice A: Glossário

Apêndice B: Modelos de análise

Apêndice C: Lista de problemas

Uma descrição detalhada de cada tópico SRS pode ser obtida fazendo-se o download da planilha SRS na URL citada anteriormente neste quadro.

Validação. Os artefatos produzidos como consequência da engenharia de requisitos são avaliados quanto à qualidade durante a etapa de validação. A validação de requisitos examina a especificação⁵ para garantir que todos os requisitos de software tenham sido declarados de for-

⁵ Lembre-se de que a natureza da especificação irá variar em cada projeto. Em alguns casos, a “especificação” é um conjunto de cenários de usuários e pouco mais do que isso. Em outros, a especificação pode ser um documento contendo cenários, modelos e descrições por escrito.



Uma preocupação fundamental durante a validação de requisitos é a consistência. Use o modelo de análise para garantir que os requisitos foram declarados de forma consistente.

ma não ambígua; que as inconsistências, omissões e erros tenham sido detectados e corrigidos e que os artefatos estejam de acordo com os padrões estabelecidos para o processo, projeto e produto.

O principal mecanismo de validação de requisitos é a revisão técnica (Capítulo 15). A equipe de revisão que valida os requisitos é formada por engenheiros de software, clientes, usuários e outros interessados que examinam a especificação em busca de erros no conteúdo ou na interpretação, áreas em que talvez sejam necessários esclarecimentos, de informações faltantes, de inconsistências (um problema grave quando são criados produtos ou sistemas grandes), de requisitos conflitantes ou de requisitos irreais (inatingíveis).

INFORMAÇÕES



Lista de controle para validação de requisitos

Muitas vezes é útil examinar cada requisito em relação a um conjunto de perguntas contidas em uma lista de controle. A seguir, um pequeno subconjunto daquelas que poderiam ser perguntadas:

- Os requisitos estão declarados de forma clara? Eles podem ser mal-interpretados?
- A fonte (por exemplo, uma pessoa, uma regulamentação, um documento) do requisito foi identificada? A declaração final do requisito foi examinada pela fonte original ou com ela?
- O requisito está limitado em termos quantitativos?
- Que outros requisitos se relacionam a este requisito? Eles estão claramente indicados por meio de uma matriz de referência cruzada ou algum outro mecanismo?
- O requisito viola quaisquer restrições do domínio do sistema?
- O requisito pode ser testado? Em caso positivo, podemos especificar testes (algumas vezes denominados critérios de validação) para testar o requisito?
- O requisito pode ser associado a qualquer modelo de sistema que tenha sido criado?
- O requisito pode ser associado aos objetivos globais do sistema/produto?
- A especificação é estruturada de forma que leve ao fácil entendimento, fácil referência e fácil tradução em artefatos mais técnicos?
- Criou-se um índice para a especificação?
- Os requisitos associados ao desempenho, ao comportamento e às características operacionais foram declarados de maneira clara? Quais requisitos parecem estar implícitos?

Gestão de requisitos. Os requisitos para sistemas baseados em computadores mudam, e o desejo de mudar os requisitos persiste ao longo da vida de um sistema. Gestão de requisitos é um conjunto de atividades que ajuda a equipe de projeto a identificar, controlar e acompanhar as necessidades e suas mudanças a qualquer momento enquanto o projeto prossegue.⁶ Muitas

FERRAMENTAS DO SOFTWARE



Engenharia de requisitos

Objetivo: As ferramentas de engenharia de requisitos auxiliam no levantamento, na modelagem, na gestão, bem como na validação de requisitos.

Mecânica: A mecânica das ferramentas varia. Em geral, as ferramentas de engenharia de requisitos constroem uma grande variedade de modelos gráficos (por exemplo, UML) que representam os aspectos informativos, funcionais e comportamentais de um sistema. Esses modelos formam a base para todas as demais atividades no processo de software.

Ferramentas representativas:⁷

Uma lista relativamente abrangente (e atualizada) de ferramentas de engenharia de requisitos pode ser encontrada no site Volvere Requirements em www.volvere.co.uk/tools.htm. As ferramentas de modelagem de requisitos são discutidas nos

Capítulos 6 e 7. As ferramentas citadas a seguir se concentram na gestão de requisitos.

EasyRM, desenvolvida pela Cybernetic Intelligence GmbH (www.easy-rm.com), constrói um dicionário/glossário específico de projetos contendo atributos e descrições detalhadas dos requisitos.

Rational RequisitePro, desenvolvida pela Rational Software (www-306.ibm.com/software/awdtools/reqpro/), permite aos usuários construir um banco de dados de requisitos, representar as relações entre requisitos e organizar, priorizar e rastrear requisitos.

Muitas outras ferramentas de gerenciamento de necessidades podem ser encontradas no site da Volvere citado anteriormente no seguinte endereço: www.jiludwig.com/Requirements_Management_Tools.html.

⁶ O gerenciamento formal de requisitos é iniciado apenas para grandes projetos com centenas de necessidades identificáveis. Para projetos pequenos, essa ação da engenharia de requisitos é consideravelmente menos formal.

⁷ As ferramentas aqui apresentadas não significam um endosso, mas sim, uma amostra de ferramentas desta categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas por seus respectivos desenvolvedores.

dessas atividades são idênticas às técnicas de gerenciamento de configurações de software (*software configuration management*, SCM) discutidas no Capítulo 22.

5.2 INÍCIO DO PROCESSO DE ENGENHARIA DE REQUISITOS

PONTO-CHAVE

Interessado é qualquer um que tenha interesse direto ou benefício-se do sistema a ser desenvolvido.

Em um ambiente ideal, os interessados e os engenheiros de software trabalham juntos na mesma equipe.⁸ Em tais casos, a engenharia de requisitos é apenas uma questão de conduzir conversações proveitosas com colegas que sejam membros bem conhecidos da equipe. Porém, a realidade muitas vezes é bastante diferente.

Cliente(s) ou usuários finais podem estar situados em uma cidade ou país diferente, podem ter apenas uma vaga ideia daquilo que é necessário, podem ter opiniões conflitantes sobre o sistema a ser construído, podem ter conhecimento técnico limitado ou quem sabe pouco tempo para interagir com o engenheiro que está fazendo o levantamento dos requisitos. Nenhuma das situações anteriores é desejável, contudo, todas são relativamente comuns e muitas vezes você é forçado a trabalhar de acordo com as limitações impostas pela situação.

Nas seções a seguir, discutiremos as etapas necessárias para estabelecer as bases para um entendimento dos requisitos de software — para que o projeto possa ser iniciado de modo que avance na direção de uma solução bem-sucedida.

5.2.1 Identificação de interessados

Sommerville e Sawyer [Som97] definem “interessados” como “qualquer um que se beneficia de forma direta ou indireta do sistema que está sendo desenvolvido”. Já tivemos a oportunidade de identificar os envolvidos usuais: gerentes de operações, gerentes de produto, pessoal de marketing, clientes internos e externos, usuários finais, consultores, engenheiros de produto, engenheiros de software, engenheiros de suporte e manutenção e outros. Cada interessado tem uma visão diferente do sistema, obtém diferentes benefícios quando o sistema é desenvolvido com êxito e está sujeito a diferentes riscos caso o trabalho de desenvolvimento venha a fracassar.

No início, devemos criar uma lista das pessoas que irão contribuir com sugestões à medida que os requisitos são obtidos (Seção 5.3). A lista inicial crescerá à medida que os interessados forem contatados, pois para cada um deles será feita a pergunta: “Com quem mais você acha que eu deva falar?”.

5.2.2 Reconhecimento de diversos pontos de vista

Pelo fato de existirem muitos interessados diferentes, os requisitos do sistema serão explorados sob vários pontos de vista. Por exemplo, o grupo de marketing está interessado nas funções e recursos que irão suscitar o mercado potencial, facilitando a venda do novo sistema. Os gerentes comerciais estão interessados em um conjunto de recursos que pode ser construído dentro do orçamento e que estarão prontos para atender às oportunidades definidas de ingresso no mercado. Os usuários finais podem querer recursos que sejam familiares a eles e que sejam fáceis de aprender e usar. Os engenheiros de software podem estar preocupados com funções invisíveis aos interessados não técnicos, mas que possibilitam uma infraestrutura que dê suporte a um maior número de funções e recursos comercializáveis. Os engenheiros de suporte talvez enfoquem a facilidade de manutenção do software.

Cada uma dessas partes envolvidas (e outras) contribuirá com informações para o processo de engenharia de requisitos. As informações dos diversos pontos de vista são coletadas, requisitos emergentes talvez sejam inconsistentes ou entrem em conflito uns com os outros. Deve-se classificar as informações de todos os interessados (inclusive os requisitos inconsistentes e conflitantes) de maneira que permita aos tomadores de decisão escolher um conjunto internamente consistente de requisitos para o sistema.

“Coloque três interessados em uma sala e pergunte a eles que tipo de sistema desejam. Provavelmente você obterá quatro ou mais opiniões diferentes.”

Autor desconhecido

8 Essa abordagem é altamente recomendada para projetos que adotam uma filosofia de desenvolvimento de software ágil.

5.2.3 Trabalho na busca da colaboração

Se cinco interessados estiverem envolvidos em um projeto de software, talvez tenhamos cinco (ou mais) opiniões diferentes sobre o conjunto de requisitos apropriado. Ao longo dos capítulos anteriores, citei que os clientes (e outros interessados) devem colaborar entre si (evitando insignificantes lutas internas pelo poder) e com os profissionais de engenharia de software, caso se queira obter um sistema bem-sucedido. Mas como a colaboração é atingida?

O trabalho de um engenheiro de requisitos é identificar áreas em comum (requisitos com os quais todos os interessados concordam) e áreas de conflito ou inconsistência (requisitos desejados por um interessado, mas que conflitam com os de outro interessado). É, obviamente, a última categoria que representa um desafio.



Utilização de “pontos de prioridade”

Um modo de resolver requisitos conflitantes e, ao mesmo tempo, entender a importância relativa de todas as necessidades é usar um esquema de “votação” baseado nos pontos de prioridade.

Todos os interessados recebem certo número de pontos de prioridade que podem ser “gastos” em um número qualquer de requisitos. É apresentada uma lista de requisitos, e cada interessado indica a importância relativa de cada um deles (sob o seu

ponto de vista) gastando um ou mais pontos de prioridade nele. Pontos gastos não podem ser reutilizados.

Uma vez que os pontos de prioridade de um interessado tenham se esgotado, nenhuma ação adicional em relação aos requisitos pode ser feita por essa pessoa. O total de pontos dados por todos os interessados a cada requisito dá uma indicação da importância global de cada requisito.

INFORMAÇÕES

Colaboração não significa necessariamente que os requisitos são definidos por um comitê. Em muitos casos, os interessados colaboram dando suas visões dos requisitos, mas um poderoso “campeão dos projetos” (por exemplo, um gerente comercial ou um técnico sênior) pode tomar a decisão final sobre quais os requisitos que passam pelo corte.

5.2.4 Perguntas iniciais

As perguntas feitas na concepção do projeto devem ser “livres de contexto” [Gau89]. O primeiro conjunto de perguntas enfoca o cliente e outros interessados, os benefícios e as metas de projeto globais. Por exemplo, poderíamos perguntar:

- Quem está por trás da solicitação deste trabalho?
- Quem irá usar a solução?
- Qual será o benefício econômico de uma solução bem-sucedida?
- Há uma outra fonte para a solução que você precisa?

Essas perguntas ajudam a identificar todos os interessados no software a ser criado. Além disso, identificam o benefício mensurável de uma implementação bem-sucedida e possíveis alternativas para o desenvolvimento de software personalizado.

O conjunto de perguntas a seguir permite que adquiramos um melhor entendimento do problema e que o cliente expresse suas percepções sobre uma solução:

- Como você caracterizaria uma “boa” saída, que seria gerada por uma solução bem-sucedida?
- Qual(is) problema(s) esta solução irá tratar?
- Você poderia me indicar (ou descrever) o ambiente de negócios em que a solução será usada?
- Restrições ou problemas de desempenho afetam a maneira com que a solução será abordada?

“É melhor conhecer algumas perguntas do que todas as respostas.”

James Thurber

? Quais perguntas irão ajudá-lo a obter um entendimento preliminar do problema?

O conjunto final de perguntas concentra-se na eficácia da atividade de comunicação em si. Gause e Weinberg [Gau89] chamam esse conjunto de “metaperguntas” e propõem a seguinte lista (sintetizada):

“Aquele que pergunta é tolo por cinco minutos; aquele que não faz é tolo para sempre.”

Provérbio chinês

- Você é a pessoa correta para responder estas perguntas? Suas respostas são “oficiais”?
- Minhas perguntas são relevantes para o problema que você tem?
- Estaria eu fazendo perguntas demais?
- Alguma outra pessoa poderia me prestar informações adicionais?
- Deveria eu perguntar-lhe algo mais?

Essas (e outras) perguntas irão ajudá-lo a “quebrar o gelo” e iniciar o processo de comunicação essencial para o êxito do levantamento. Porém, uma reunião com formato perguntas-e-respostas não é uma abordagem que tem obtido um sucesso marcante. Na realidade, a sessão Q&A (perguntas e respostas) deveria ser usada apenas no primeiro encontro e depois substituída pelo formato de levantamento de requisitos que combina elementos de resolução de problemas, negociação e especificação. Uma abordagem desse tipo é apresentada na Seção 5.3.

5.3 LEVANTAMENTO DE REQUISITOS

O levantamento de requisitos (também chamado *elicitação de requisitos*) combina elementos de resolução de problemas, elaboração, negociação e especificação. Para encorajar uma abordagem colaborativa e orientada às equipes em relação ao levantamento de requisitos, os interessados trabalham juntos para identificar o problema, propor elementos da solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos da solução [Zah90].⁹

5.3.1 Coleta colaborativa de requisitos

Foram propostas várias abordagens para a coleta colaborativa de requisitos. Cada uma delas faz uso de um cenário ligeiramente diferente, porém todas aplicam alguma variação das seguintes diretrizes básicas:

 **Quais são as diretrizes básicas para conduzir uma reunião de coleta colaborativa de requisitos?**

- As reuniões são conduzidas por e com a participação tanto dos engenheiros de software quanto de outros interessados.
- São estabelecidas regras para preparação e participação.
- É sugerida uma agenda suficientemente formal para cobrir todos os pontos importantes, porém, suficientemente informal para encorajar o fluxo livre de ideias.
- Um “facilitador” (pode ser um cliente, um desenvolvedor ou uma pessoa de fora) dirige a reunião.
- É utilizado um “mecanismo de definições” (que pode ser planilhas, *flip charts*, adesivos de parede ou um boletim eletrônico, salas de bate-papo ou fóruns virtuais).

“Gastamos um bom tempo — a maior parte do esforço de um projeto — não implementando ou testando, mas sim tentando decidir o que construir.”

Brian Lawrence

A meta é identificar o problema, propor elementos da solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos da solução em uma atmosfera que seja propícia para o cumprimento da meta. Para melhor compreender o fluxo de eventos à medida que ocorrem, apresento um breve cenário que descreve em linhas gerais a sequência de eventos que levam à reunião para levantamento de requisitos e que acontecem durante e após a reunião.

Durante a concepção (Seção 5.2) perguntas e respostas estabelecem o escopo do problema e a percepção geral de uma solução. Como resultado dessas reuniões iniciais, o desenvolvedor e os clientes redigem uma “solicitação de produto” de uma ou duas páginas.

⁹ Essa abordagem é algumas vezes chamada FAST (*facilitated application specification technique*, ou seja, técnica de especificação de aplicações facilitada).

WebRef

Joint Application Development (JAD) é uma popular técnica para levantamento de requisitos. Uma excelente descrição sobre ela pode ser encontrada em www.carolla.com/wp-jad.htm.



Se um sistema ou produto é atender muitos usuários, esteja absolutamente certo de que os requisitos foram extraídos de uma fatia representativa dos usuários. Se apenas um usuário definiu todas as necessidades, o risco de não aceitação é grande.

"Fatos não cessam de existir por serem ignorados."

Aldous Huxley



Evite o impulso de hostilizar uma sugestão de um cliente classificando-a como "muito cara" ou "pouco prática". O objetivo aqui é negociar uma lista que seja aceitável para todos. Para tanto, você deve ser receptivo a novas ideias.

São escolhidos um local, hora e data para a reunião; é escolhido um facilitador; e os membros da equipe de software e de outros departamentos interessados são convidados a participar. A solicitação de produto é distribuída a todos os participantes antes da data da reunião.

Como exemplo,¹⁰ consideremos um trecho de uma solicitação de produto redigida por uma pessoa de marketing envolvida no projeto *CasaSegura*. Esta pessoa escreve a seguinte narrativa sobre a função de segurança domiciliar que faz parte do *CasaSegura*:

Nossa pesquisa indica que o mercado para sistemas de administração domiciliar está crescendo com taxas de 40% ao ano. A primeira função do *CasaSegura* que lançaríamos no mercado seria a função de segurança domiciliar. A maioria das pessoas está familiarizada com "sistemas de alarme", de modo que seria algo fácil de vender.

A função de segurança domiciliar protegeria e/ou reconheceria uma série de "situações" indesejáveis como acesso ilegal, incêndios, inundações, níveis de monóxido de carbono e outras. Ela usará nossos sensores sem fio para detectar cada uma dessas situações. Poderá ser programada pelo proprietário da casa e, automaticamente, ligará para uma agência de monitoramento quando for detectada uma situação destas.

Na realidade, outras pessoas contribuiriam para essa narrativa durante a reunião para levantamento de requisitos e um número consideravelmente maior de informações ficariam disponíveis. Contudo, mesmo com essas informações adicionais, a ambiguidade estaria presente, provavelmente existiriam omissões e poderiam ocorrer erros. Por enquanto, a "descrição funcional" anterior será suficiente.

Ao rever a solicitação de produto nos dias que antecedem a reunião, é pedido a cada participante uma lista de objetos que fazem parte do ambiente que cerca o sistema, outros que devem ser produzidos pelo sistema e aqueles usados pelo sistema para desempenhar suas funções. Além disso, pede-se a cada participante uma outra lista de serviços (processos ou funções) que manipulam ou interagem com os objetos. Por fim, também são desenvolvidas listas de restrições (por exemplo, custo, dimensões, regras comerciais) e de critérios de desempenho (por exemplo, velocidade, precisão). Os participantes são informados que não se espera que as listas sejam exaustivas, mas que reflitam a percepção do sistema de cada pessoa.

Entre os objetos descritos para o *CasaSegura* poderíamos ter o painel de controle, detectores de fumaça, sensores para janelas e portas, detectores de movimento, um alarme, um evento (por exemplo, um sensor foi ativado), um display, um PC, números de telefone, uma ligação telefônica e assim por diante. A lista de serviços poderia incluir *configurar* o sistema, *acionar* o alarme, *monitorar* os sensores, *discar* o telefone, *programar* o painel de controle e *ler* o display (note que os serviços atuam sobre os objetos). De maneira similar, cada participante irá criar listas de restrições (por exemplo, o sistema tem de reconhecer quando os sensores não estão operando, ser amigável, interfacear diretamente com uma linha telefônica comum) e de critérios de desempenho (por exemplo, um evento de sensor seria reconhecido em um intervalo de um segundo e seria implementado um esquema de prioridade para os eventos).

As listas de objetos poderiam ser afixadas nas paredes da sala de reunião utilizando-se grandes folhas de papel, coladas nas paredes por meio de folhas adesivas ou escritas em mural. Como alternativa, poderiam ser postadas em um boletim eletrônico, em um site interno ou colocadas em um ambiente de salas de bate-papo para revisão antes da reunião. De modo ideal, cada entrada deveria ser capaz de ser manipulada separadamente, de modo que as listas pudessem ser combinadas, as entradas modificadas e adições feitas. Nesse estágio, críticas e polêmicas são estritamente proibidas.

¹⁰ Esse exemplo (com extensões e variações) é usado para ilustrar importantes métodos de engenharia de software em vários dos capítulos que vêm a seguir. Como exercício, valeria a pena realizar sua própria reunião para levantamento de requisitos e desenvolver um conjunto de listas para ela.

Depois de as listas individuais terem sido apresentadas, o grupo cria uma lista combinada eliminando entradas redundantes, acrescentando quaisquer ideias novas que surjam durante a discussão, mas não se elimina nada. Segue-se então uma discussão (coordenada pelo facilitador). A lista combinada é reduzida, ampliada ou escrita de outra maneira para refletir apropriadamente o produto/sistema a ser desenvolvido. O objetivo é criar uma lista consensual de objetos, serviços, restrições e desempenho para o sistema a ser construído.

Em muitos casos, um objeto ou serviço descrito em uma lista exigirá maiores explicações. Para tanto, os interessados desenvolvem *miniespecificações* para as entradas das listas.¹¹ Cada miniespecificação é a elaboração de um objeto ou serviço. Por exemplo, a miniespecificação para o objeto **Control Panel** do *CasaSegura* poderia ser:

O painel de controle é a unidade que pode ser montada na parede com tamanho aproximado de 9 x 5 polegadas. O painel de controle possui conectividade sem fio a sensores e a um PC. A interação com os usuários ocorre com um teclado numérico contendo 12 teclas. Um display colorido LCD de 3 x 3 polegadas fornece o feedback do usuário. O software fornece prompts interativos, eco e funções similares.

As miniespecificações são apresentadas a todos os interessados para discussão. Adições, supressões e mais detalhamentos são feitos. Em alguns casos, o desenvolvimento de miniespecificações irá revelar novos objetos, serviços, restrições, ou requisitos de desempenho acrescentados às listas originais. Durante todas as discussões, a equipe pode levantar um assunto que não pode ser resolvido durante a reunião. É mantida uma *lista de questões pendentes* para que essas ideias sejam trabalhadas posteriormente.

CASA SEGURA



Condução de uma reunião para levantamento de requisitos

Cena: Uma sala de reunião. A primeira reunião para levantamento de requisitos está em andamento.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto e um facilitador.

Conversa:

Facilitador (apontando para uma lousa branca): Portanto, esta é a lista atual de objetos e serviços para a função segurança domiciliar.

Representante do Depto. de Marketing:

Segundo o nosso ponto de vista, esta lista cobre aproximadamente todas as funcionalidades.

Vinod: Alguém não mencionou que eles queriam que toda a funcionalidade do CasaSegura pudesse ser acessada via Internet?

Isso incluiria a função de segurança domiciliar, não é mesmo?

Representante do Depto. de Marketing: Sim, você está certo... Teremos de acrescentar essa funcionalidade e os objetos apropriados.

Facilitador: Isso também não acrescentaria algumas restrições?

Jamie: Realmente, restrições técnicas e legais.

Representante da Engenharia de Produto: E isso significa o quê?

Jamie: É melhor termos certeza de que um intruso não conseguirá entrar no sistema, desarmá-lo e roubar o local ou coisa pior. Grande responsabilidade sobre nós.

Doug: Uma grande verdade.

Representante do Depto. de Marketing: Mas ainda assim precisamos que... Apenas ter certeza de impedir que um intruso entre.

Ed: É fácil dizer, o duro é fazer...

Facilitador (interrompendo): Não gostaria de discutir esta questão agora. Anotemos a questão para ser trabalhada no futuro e prossigamos.

(Doug, atuando como o secretário da reunião, faz um apontamento apropriado.)

Facilitador: Tenho a impressão de que ainda há mais coisas a ser consideradas aqui.

(O grupo gasta os 20 minutos seguintes refinando e expandindo os detalhes da função segurança domiciliar.)

¹¹ Em vez de criar uma miniespecificação, muitas equipes de software optam por desenvolver cenários de usuários denominados *casos de uso*. Estes são considerados de forma detalhada na Seção 5.4 e no Capítulo 6.

PONTO-CHAVE

O QFD define necessidades para maximizar a satisfação do cliente.



Todo mundo quer implementar um monte de requisitos fascinantes, porém, tenha cuidado. É assim que o “surgimento incontrolado de novos requisitos” se estabelece. Por outro lado, requisitos fascinantes levam a um produto revolucionário!

WebRef

Úteis informações sobre o QFD podem ser obtidas em www.qfdi.org.

5.3.2 Disponibilização da função de qualidade

A disponibilização da função de qualidade (*quality function deployment*, QFD) é uma técnica de gestão da qualidade que traduz as necessidades do cliente para requisitos técnicos do software. O QFD “concentra-se em maximizar a satisfação do cliente por meio do processo de engenharia de software” [Zul92]. Para tanto, enfatiza o entendimento daquilo que é valioso para o cliente e emprega esses valores ao longo do processo de engenharia. O QFD identifica três tipos de necessidades [Zul92]:

Requisitos normais. Refletem os objetivos e metas estabelecidos para um produto ou sistema durante reuniões com o cliente. Se esses requisitos estiverem presentes, o cliente fica satisfeito. Exemplos de requisitos normais poderiam ser tipos de displays gráficos solicitados, funções de sistema específicas e níveis de desempenho definidos.

Requisitos esperados. Esses requisitos estão implícitos no produto ou sistema e podem ser tão fundamentais que o cliente não os declara explicitamente. Sua ausência será causa de grande insatisfação. Exemplos de requisitos esperados são: facilidade na interação homem-máquina, confiabilidade e correção operacional global e facilidade na instalação do software.

Requisitos fascinantes. Esses recursos vão além da expectativa dos clientes e demonstram ser muito satisfatórios quando presentes. Por exemplo, o software para um novo celular vem com recursos-padrão, mas junto vem um conjunto de capacidades não esperadas (por exemplo, tecla multitoque, correio de voz visual) que deleitam todos os usuários do produto.

Embora os conceitos QFD possam ser aplicados ao longo de todo o processo de software [Par 96a], técnicas QFD específicas são aplicáveis à atividade de levantamento de requisitos. O QFD usa observação e entrevistas com clientes, pesquisas e exame de dados históricos (por exemplo, relatórios de problemas) como dados brutos para a atividade de levantamento de requisitos. Esses dados são então traduzidos em uma tabela de requisitos — denominada *tabela da voz do cliente* — revisada com o cliente e outros interessados. Uma série de diagramas, matrizes e métodos de avaliação é então usada para extrair os requisitos esperados e para tentar obter os requisitos fascinantes [Aka04].

5.3.3 Cenários de uso

À medida que os requisitos são levantados, uma visão geral das funções e características começa a se materializar. Entretanto, é difícil progredir para atividades de engenharia de software mais técnicas até que entendamos como tais funções e características serão usadas por diferentes classes de usuários. Para tanto, os desenvolvedores e usuários podem criar um conjunto de cenários que identifique um roteiro de uso para o sistema a ser construído. Os cenários, normalmente chamados *casos de uso* [Jac92], fornecem uma descrição de como o sistema será utilizado. Casos de uso são discutidos de forma mais detalhada na Seção 5.4.

CASASEGURA



Desenvolvimento de um cenário de uso preliminar

Cena: Sala de reuniões, na qual prossegue a primeira reunião de levantamento de requisitos.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de soft-

ware; três membros do Depto. de Marketing; um representante da Engenharia de Produto e um facilitador.

Conversa:

Facilitador: Andamos conversando sobre segurança de acesso à funcionalidade CasaSegura que poderá ser acessada via Internet. Gostaria de tentar algo. Vamos desenvolver um cenário de uso para acesso à função segurança domiciliar.

Jamie: Como?

Facilitador: Podemos realizar isso de várias maneiras, mas, por enquanto, gostaria de manter as coisas realmente informais. Conte-nos (ele aponta para um representante do Depto. de Marketing) como você imagina o acesso ao sistema.

Representante do Depto. de Marketing: Huum... Bem, esse é o tipo de coisa que eu faria se estivesse fora e tivesse que deixar alguém em casa, digamos uma empregada ou um encanador, que não teriam o código de acesso.

Facilitador (sorrindo): Isso que você falou é a razão para você fazê-lo... Diga-me como realmente faria isso.

Representante do Depto. de Marketing: Huum... A primeira coisa de que precisaria seria um PC. Entraria em um site que deveríamos manter para todos os usuários do CasaSegura. Forneceria meu nome de usuário e...

Vinod (interrompendo): A página teria de ser segura e criptografada para garantir que estamos seguros e...

Facilitador (interrompendo): Essas são informações adequadas, Vinod, porém são técnicas. Concentremo-nos apenas em como o usuário final usará essa capacidade. OK?

Vinod: Sem problemas.

Representante do Depto. de Marketing: Bem, como estava dizendo, entraria em um site e forneceria meu nome de usuário e dois níveis de senhas.

Jamie: O que acontece se eu esquecer minha senha?

Facilitador (interrompendo): Excelente observação, Jamie, mas não tratemos disso agora. Faremos um registro de sua observação e a chamaremos de exceção. Tenho certeza de que existirão outras.

Representante do Depto. de Marketing: Após eu introduzir as senhas, uma tela representando todas as funções do CasaSegura aparecerá. Eu selecionaria a função de segurança domiciliar. O sistema poderia solicitar que eu verificasse quem sou eu, digamos, perguntando meu endereço ou telefone ou algo do gênero. Em seguida, ele exibiria uma imagem do painel de controle do sistema de segurança com uma lista das funções que eu poderia executar — armar o sistema, desarmá-lo, desarmar um ou mais sensores. Suponho que ele também me permitiria reconfigurar zonas de segurança e outros itens similares, mas, não tenho certeza disso.

(Enquanto o representante do Depto. de Marketing continua falando, Doug faz anotações de maneira ininterrupta; essas formam a base para o primeiro cenário de uso informal. Como alternativa, poderia ser solicitado ao representante do Depto. de Marketing que escrevesse o cenário, mas isso seria feito fora da reunião.)

5.3.4 Artefatos do levantamento de requisitos

Os artefatos produzidos como consequência do levantamento de requisitos variarão dependendo do tamanho do sistema ou produto a ser construído. Para a maioria dos sistemas, entre os artefatos, temos:

 **Quais informações são produzidas como consequência do levantamento de requisitos?**

- Uma declaração da necessidade e viabilidade.
- Uma declaração restrita do escopo para o sistema ou produto.
- Uma lista de clientes, usuários e outros interessados que participaram do levantamento de requisitos.
- Uma descrição do ambiente técnico do sistema.
- Uma lista de requisitos (preferencialmente organizada por função) e as restrições de domínio que se aplicam a cada uma delas.
- Um conjunto de cenários de uso que esclarecem o uso do sistema ou produto sob diferentes condições operacionais.
- Quaisquer protótipos desenvolvidos para melhor definição dos requisitos.

Cada um desses artefatos é revisado por todas as pessoas que participaram do levantamento de requisitos.

5.4 DESENVOLVIMENTO DE CASOS DE USO

Em um livro que discute como redigir casos de uso eficazes, Alistair Cockburn [Coc01b] observa que “um caso de uso captura um contrato... [que] descreve o comportamento do sistema sob várias condições à medida que o sistema responde a uma solicitação de um de seus interessados...”. Essencialmente, um caso de uso conta uma história estilizada sobre como um usuário final (desempenhando um de uma série de papéis possíveis) interage com o sistema sob um

PONTO-CHAVE

Os casos de uso são definidos sob o ponto de vista de um ator. Ator é um papel que as pessoas (usuários) ou dispositivos desempenham à medida que interagem com o software.

WebRef

Um excelente artigo sobre casos de uso pode ser baixado de www.ibm.com/developerworks/webservices/library/codesign7.html.

 O que preciso saber para desenvolver um caso de uso efetivo?

conjunto de circunstâncias específicas. A história poderia ser um texto narrativo, uma descrição geral das tarefas ou interações, uma descrição baseada em gabaritos ou uma representação esquemática. Independentemente de sua forma, um caso de uso representa o software ou o sistema do ponto de vista do usuário final.

O primeiro passo ao escrever um caso de uso é definir o conjunto de “atores” envolvidos na história. *Atores* são as diferentes pessoas (ou dispositivos) que usam o sistema ou produto no contexto da função e comportamento a ser descritos. Os atores representam os papéis que pessoas (ou dispositivos) desempenham enquanto o sistema opera. Definido de maneira um pouco mais formal, ator é qualquer coisa que se comunica com o sistema ou o produto e que é externa ao sistema em si. Todo ator possui uma ou mais metas ao usar o sistema.

É importante notar que o ator e o usuário final não são necessariamente a mesma coisa. O usuário típico poderia desempenhar uma série de papéis diferentes ao usar um sistema, ao passo que o ator representa uma classe de entidades externas (normalmente, mas não sempre, pessoas) que desempenham apenas um papel no contexto do caso de uso. Como exemplo, consideremos um operador de máquina (um usuário) que interage com o computador de controle de uma célula de fabricação contendo uma série de robôs e máquinas comandada por controle numérico. Após uma revisão cuidadosa dos requisitos, o software para o computador de controle requer quatro modos (papéis) diferentes para interação: modo de programação, modo de teste, modo de monitoramento e modo de diagnóstico. Portanto, podem ser definidos quatro atores: programador, testador, monitorador e diagnosticador. Em alguns casos, o operador de máquina pode desempenhar todos esses papéis. Em outros, pessoas diferentes poderiam desempenhar o papel de cada ator.

Pelo fato de o levantamento de requisitos ser uma atividade evolucionária, nem todos os atores são identificados durante a primeira iteração. É possível identificar atores primários [Jac92] durante a primeira iteração e atores secundários quando mais fatos são aprendidos sobre o sistema. Os *primários* interagem para atingir a função necessária do sistema e obter o benefício desejado do sistema. Eles trabalham direta e frequentemente com o software. Os *secundários* dão suporte ao sistema, de modo que os primários possam realizar seu trabalho.

Uma vez que os atores tenham sido identificados, os casos de uso podem ser desenvolvidos. Jacobson [Jac92] sugere uma série de perguntas¹² que devem ser respondidas por um caso de uso:

- Quem é(são) o(s) ator(es) primário(s) e o(s) ator(es) secundário(s)?
- Quais são as metas do ator?
- Que condições devem existir antes de uma história começar?
- Que tarefas ou funções principais são realizadas pelo ator?
- Que exceções deveriam ser consideradas à medida que uma história é descrita?
- Quais são as variações possíveis na interação do ator?
- Que informações de sistema o ator adquire, produz ou modifica?
- O ator terá de informar o sistema sobre mudanças no ambiente externo?
- Que informações o ator deseja do sistema?
- O ator gostaria de ser informado sobre mudanças inesperadas?

Relembrando as necessidades básicas do *CasaSegura*, definimos quatro atores: **proprietário** (um usuário), **gerente de ativação** (provavelmente a mesma pessoa que o **proprietário**, porém desempenhando um papel diferente), **sensores** (dispositivos conectados ao sistema) e o **subsistema de monitoramento e resposta** (a estação central que monitora a função segurança domiciliar do *CasaSegura*). Para o propósito deste exemplo, consideraremos apenas o ator **proprietário**, que interage com a função segurança domiciliar de uma de várias maneiras diferentes usando o painel de controle de alarme ou um PC:

¹² As perguntas de Jacobson foram estendidas para dar uma visão mais completa do conteúdo dos casos de uso.

- Digita uma senha para permitir todas as demais interações.
- Consulta o estado de uma zona de segurança.
- Consulta o estado de um sensor.
- Pressiona o botão de alarme em caso de emergência.
- Ativa/desativa o sistema de segurança.

Considerando-se a situação em que o proprietário do imóvel usa o painel de controle, o caso de uso básico para ativação do sistema é o seguinte:¹³

1. O proprietário observa o painel de controle do *CasaSegura* (Figure 5.1) para determinar se o sistema está pronto para entrada. Se o sistema não estiver pronto, será mostrada uma mensagem *não disponível* no display LCD e o proprietário terá de fechar manualmente as janelas ou portas para que a mensagem *não disponível* desapareça. [Uma mensagem *não disponível* implica que um sensor está aberto; isto é, que uma porta ou janela está aberta.]
2. O proprietário usa o teclado numérico para introduzir uma senha de quatro dígitos. A senha é comparada com a senha válida armazenada no sistema. Se estiver incorreta, o painel de controle emitirá um bipe uma vez e se autorreiniciará na espera de entrada adicional. Se a senha estiver correta, o painel de controle aguarda por novas ações.
3. O proprietário seleciona e digita *em casa* ou *fora de casa* (veja a Figura 5.1) para ativar o sistema. *Em casa* ativa apenas sensores periféricos (os sensores para detecção de movimento interno são desativados). *Fora de casa* ativa todos os sensores.
4. Quando ocorre a ativação, uma luz de alarme vermelha pode ser observada pelo proprietário.



Os casos de uso são normalmente escritos de forma informal. Entretanto, use o modelo aqui mostrado para garantir que você tratou todas as questões-chave.

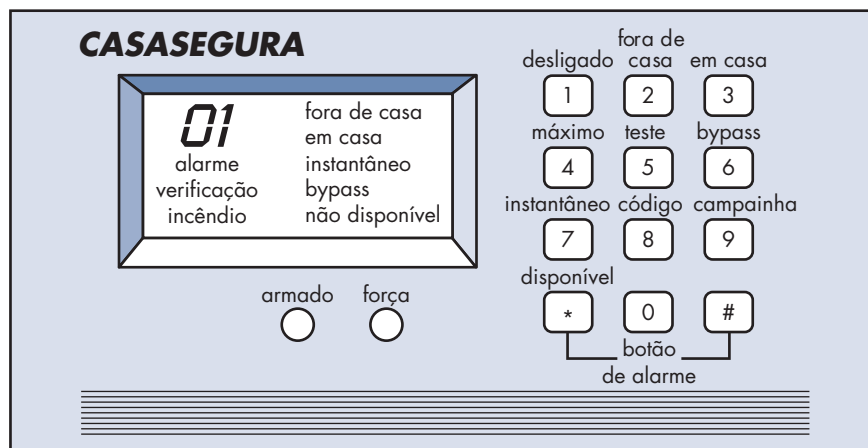
O caso de uso básico apresenta uma história detalhada que descreve a interação entre o ator e o sistema.

Em muitas ocasiões, os casos de uso são mais elaborados para dar um nível de detalhes consideravelmente maior sobre a interação. Por exemplo, Cockburn [Coc01b] sugere o seguinte modelo para descrições detalhadas de casos de uso:

Caso de uso:	<i>IniciarMonitoramento</i>
Ator primário:	Proprietário.
Meta do contexto:	Ativar o sistema para monitoramento dos sensores quando o proprietário deixa a casa ou nela permanece.

FIGURA 5.1

Painel de controle do CasaSegura



¹³ Note que esse caso de uso difere da situação em que o sistema é acessado via Internet. Nessa situação, a interação ocorre através do painel de controle e não da interface gráfica do usuário (GUI) fornecida quando é utilizado um PC.

Precondições:	O sistema foi programado para uma senha e para reconhecer vários sensores.
Disparador:	O proprietário decide “acionar” o sistema, isto é, ativar as funções de alarme.

Cenário:

1. Proprietário: observa o painel de controle
2. Proprietário: introduz a senha
3. Proprietário: seleciona “em casa” ou “fora de casa”
4. Proprietário: observa a luz de alarme vermelha para indicar que o CasaSegura foi armado

Exceções:

1. O painel de controle encontra-se no estado *não disponível*: o proprietário verifica todos os sensores para determinar quais estão abertos; fechando-os.
2. A senha incorreta (o painel de controle emite um bipe): o proprietário introduz novamente a senha, desta vez correta.
3. Senha não reconhecida: o subsistema de monitoramento e resposta deve ser contatado para reprogramar a senha.
4. É selecionado *em casa*: o painel de controle emite dois bipes e uma luz de *em casa* é acesa; os sensores periféricos são ativados.
5. É selecionado *fora de casa*: o painel de controle emite três bipes e uma luz *fora de casa* é acesa; todos os sensores são ativados.

Prioridade:	Essencial, deve ser implementada
Quando disponível:	Primeiro incremento
Frequência de uso:	Várias vezes por dia
Canal com o ator:	Via interface do painel de controle
Atores secundários:	Técnico de suporte, sensores
Canais com os atores secundários:	
	Técnico de suporte: linha telefônica
	Sensores: interfaces <i>hardwired</i> e de radiofrequência

Questões em aberto:

1. Existiria um modo de ativar o sistema sem o uso de uma senha ou com uma senha abreviada?
2. Deveria o painel de controle exibir outras mensagens de texto?
3. Quanto tempo o proprietário tem para introduzir a senha a partir do instante em que a primeira tecla é pressionada?
4. Existe alguma maneira de desativar o sistema antes de ser realmente ativado?

Casos de uso para outras interações do **proprietário** seriam desenvolvidos de maneira similar. É importante revisar cada caso com cuidado. Se algum elemento da interação for ambíguo, é provável que uma revisão do caso de uso irá indicar um problema.

CASA SEGURA**Desenvolvimento de um diagrama de caso de uso detalhado**

Cena: Sala de reuniões, na qual prossegue a reunião para levantamento de necessidades.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da

equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto e um facilitador.

Conversa:

Facilitador: Investimos um tempo razoável falando sobre a funcionalidade de segurança domiciliar do CasaSegura. Durante o intervalo esbocei um diagrama de caso de uso para sintetizar

os cenários importantes que fazem parte desta função. Deem uma olhada.

(Todos os participantes observam a Figura 5.2.)

Jamie: Estou apenas começando a aprender a notação UML.¹⁴ Portanto, a função de segurança domiciliar é representada pelo retângulo grande com as elipses em seu interior? E as elipses representam os casos de uso que redigimos?

Facilitador: Certo. E as figuras de bonecos representam atores — as pessoas ou coisas que interagem com o sistema conforme descrito pelo caso de uso... Oh, eu uso o quadrado legendado para representar um ator que não é uma pessoa... Neste caso, sensores.

Doug: Isso é permitido na UML?

Facilitador: Permissão não é o problema. O ponto é comunicar a informação. Eu vejo o uso de uma figura de boneco para representar um dispositivo enganoso. Portanto, adaptei um pouco as coisas. Não acho que isso irá criar problemas.

Vinod: OK, temos narrativas de casos de uso para cada uma das elipses. Precisamos desenvolver narrativas baseadas nos modelos detalhados sobre os quais li a respeito?

Facilitador: Provavelmente, mas isso pode esperar até que tenhamos considerado outras funções do CasaSegura.

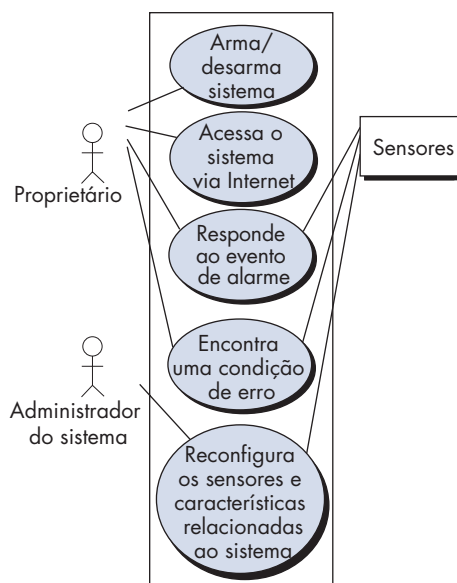
Representante do Depto. de Marketing: Espere um pouco. Fiquei observando este diagrama e de repente me dei conta de que deixamos passar algo.

Facilitador: Ah é? Diga-me o que deixamos passar.

(A reunião continua.)

FIGURA 5.2

Diagrama de caso de uso em UML para a função de segurança domiciliar do CasaSegura



FERRAMENTAS DO SOFTWARE



Desenvolvimento de caso de uso

Objetivo: Auxiliar no desenvolvimento de casos de uso através de modelos e mecanismos automatizados para avaliar a clareza e a consistência.

Mecânica: A mecânica das ferramentas varia. Em geral, as ferramentas de caso de uso fornecem modelos em que se pode preencher os espaços em branco para criar casos de uso efetivos. A maior parte da funcionalidade de caso de uso está embutida em um conjunto de funções de engenharia de requisitos mais abrangentes.

Ferramentas representativas:¹⁵

A grande maioria das ferramentas de modelagem de análise baseadas em UML oferece recursos de texto como gráficos para o desenvolvimento e a modelagem de casos de uso.

Objects by Design

(www.objectsbydesign.com/tools/umltools_byCompany.html) oferece um grande número de links para ferramentas desse tipo.

¹⁴ Um breve tutorial sobre a UML é apresentado no Apêndice 1 para aqueles que não estão familiarizados com sua notação.

¹⁵ As ferramentas aqui apresentadas não significam um aval, mas sim, uma amostra de ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

5.5 CONSTRUÇÃO DO MODELO DE ANÁLISE¹⁶

O intuito do modelo de análise é fornecer uma descrição dos domínios de informação, funcional e comportamental necessários para um sistema baseado em computadores. O modelo se modifica dinamicamente à medida que você aprende mais sobre o sistema a ser construído e outros interessados adquirem um melhor entendimento sobre aquilo que eles realmente querem. Por essa razão, o modelo de análise é uma reprodução das necessidades em determinado momento. Você deve esperar que ele sofra mudanças.

À medida que o modelo de requisitos evolui, certos elementos se tornarão relativamente estáveis, fornecendo uma sólida base para as tarefas posteriores do projeto. Entretanto, outros elementos do modelo podem ser mais voláteis, indicando que os interessados ainda não têm um entendimento completo das necessidades para o sistema. O modelo de análise e os métodos usados para construí-lo são apresentados em detalhe nos Capítulos 6 e 7. Apresento uma breve visão geral nas seções a seguir.

5.5.1 Elementos do modelo de análise

Há várias maneiras de examinar os requisitos para um sistema baseado em computador. Alguns profissionais de software argumentam que é melhor selecionar um modo de representação (por exemplo, o caso de uso) e aplicá-lo em detrimento de todos os demais. Outros profissionais acreditam que vale a pena usar uma série de modos de representação para representar o modelo de requisitos. Modos de representação diferentes nos forçam a considerar as necessidades de diferentes pontos de vista — uma abordagem com maior probabilidade de revelar omissões, inconsistências e ambiguidades.

Os elementos específicos do modelo de análise são ditados pelo método de modelagem de análise (Capítulos 6 e 7) que será usado. Entretanto, um conjunto de elementos genéricos é comum à maioria dos modelos de análise.



É sempre uma boa ideia fazer com que os interessados se envolvam. Uma das melhores formas para tal é pedir a cada interessado que escreva casos de uso que descrevam como o software será utilizado.



Uma maneira de isolar classes é procurar substantivos descritivos em um texto de caso de uso. Pelo menos alguns dos substantivos serão candidatos a classes. Mais sobre isso no Capítulo 8.

Elementos baseados em cenários. O sistema é descrito sob o ponto de vista do usuário usando uma abordagem baseada em cenários. Por exemplo, casos de uso básicos (Seção 5.4) e seus diagramas de casos de uso correspondentes (Figura 5.2) evolui para casos de uso mais elaborados baseados em modelos. Elementos do modelo de requisitos baseado em cenários são em geral a primeira parte do modelo a ser desenvolvida. Como tal, servem como entrada para a criação de outros elementos de modelagem. A Figura 5.3 mostra um diagrama¹⁷ de atividades em UML para o levantamento de requisitos e os representa utilizando casos de uso. São mostrados três níveis da elaboração, culminando em uma representação baseada em cenários.

Elementos baseados em classes. Cada cenário de uso implica um conjunto de objetos manipulados à medida que um ator interage com o sistema. Esses objetos são categorizados em classes — um conjunto de coisas que possuem atributos similares e comportamentos comuns. Por exemplo, um diagrama de classes UML pode ser utilizado para representar uma classe **Sensor** para a função de segurança do *CasaSegura* (Figura 5.4). Note que o diagrama enumera os atributos dos sensores (por exemplo, nome, tipo) e as operações (por exemplo, *identificar*, *habilitar*) que podem ser aplicadas para modificar tais atributos. Além dos diagramas de classes, outros elementos de modelagem de análise descrevem o modo pelo qual as classes colaboram entre si e os relacionamentos e interações entre as classes. Estes são discutidos de forma mais detalhada no Capítulo 7.

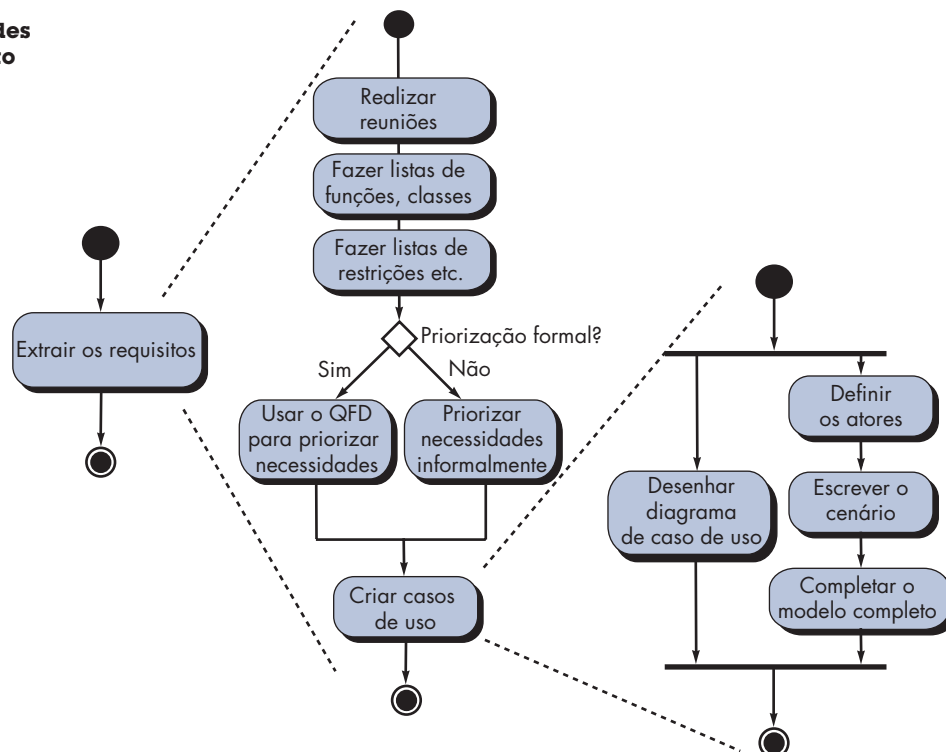
Elementos comportamentais. O comportamento de um sistema baseado em computadores pode ter um efeito profundo sobre o projeto escolhido e a abordagem de implementação

¹⁶ Ao longo deste livro, uso os termos *modelo de análise* e *modelo de requisito* como sinônimos. Ambos se referem às representações dos domínios de informação, funcional e comportamental que descrevem as necessidades dos problemas.

¹⁷ Um breve tutorial sobre a UML é apresentado no Apêndice 1 para aqueles que não estão familiarizados com sua notação.

FIGURA 5.3

Diagramas de atividades UML para levantamento de requisitos



aplicada. Portanto, o modelo de análise deve fornecer elementos de modelagem que descrevem comportamento.

PONTO-CHAVE

Um estado é um modo de comportamento externamente observável. Estímulos externos provocam transições entre estados.

O *diagrama de estados* é um método para representar o comportamento de um sistema através da representação de seus estados e dos eventos que fazem com que o sistema mude de estado. *Estado* é qualquer modo de comportamento externamente observável. Além disso, o diagrama de estados indica ações (por exemplo, ativação de processos) tomadas em decorrência de determinado evento.

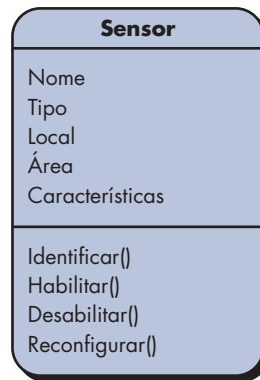
Para ilustrar o uso de um diagrama de estados, considere o software embutido no painel de controle do *CasaSegura* responsável pela leitura das entradas feitas pelos usuários. A Figura 5.5 mostra um diagrama de estados UML simplificado.

Além das representações comportamentais do sistema como um todo, o comportamento de classes individuais também pode ser modelado. Uma discussão mais aprofundada de modelagem comportamental é apresentada no Capítulo 7.

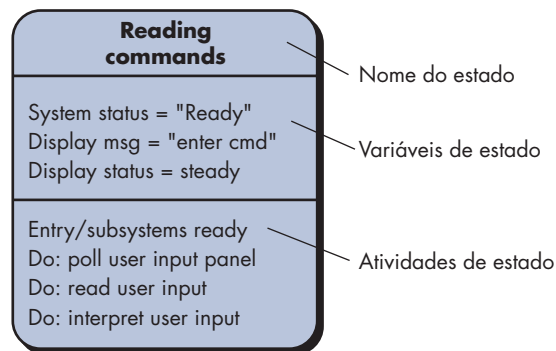
Elementos orientados a fluxos. Informações são transformadas à medida que fluem através de um sistema baseado em computador. O sistema aceita entrada em uma variedade de formas, aplica funções para transformá-las e gera saída também em uma variedade de formas. A entrada pode ser um sinal de controle transmitido por um transdutor, uma série de números digitados por um operador, um pacote de informações transmitido em um link de rede ou um arquivo de dados volumoso recuperado de armazenamento secundário. A(s) transformação(ões) pode(m) compreender desde uma única comparação lógica, um algoritmo numérico complexo até uma abordagem por inferência de regras de um sistema especialista. A saída poderia acender um único LED ou gerar um relatório de 200 páginas. Na realidade, podemos criar um modelo de fluxos para qualquer sistema baseado em computadores, indiferentemente de seu tamanho e complexidade. Uma discussão mais detalhada da modelagem de fluxos é apresentada no Capítulo 7.

FIGURA 5.4

Diagrama de classes para Sensor

**FIGURA 5.5**

Notação de um diagrama de estados UML



CASA SEGURA



Modelagem comportamental inicial

Cena: Sala de reuniões, na qual prossegue a primeira reunião para levantamento de re-

quisitos.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto e um facilitador.

Conversa:

Facilitador: Estamos prestes a finalizar nossa discussão sobre a funcionalidade segurança domiciliar do CasaSegura. Antes de fazê-lo, gostaria de discutir o comportamento da função.

Representante do Depto. de Marketing: Não entendi o que você quis dizer com comportamento.

Ed (sorrindo): Trata-se de dar ao produto um "tempo de espera" caso ele se comporte mal.

Facilitador: Não exatamente. Permita-me explicar.

(O facilitador explica os fundamentos da modelagem comportamental à equipe de levantamento de requisitos.)

Representante do Depto. de Marketing: Isso me parece um tanto técnico. Não estou certo se serei de alguma ajuda aqui.

Facilitador: Certamente que você pode ajudar. Que comportamento você observa do ponto de vista de usuário?

Representante do Depto. de Marketing: Bem, o sistema estará monitorando os sensores. Lendo comandos do proprietário. Mostrando seu estado.

Facilitador: Viu, você pode fazê-lo.

Jamie: Ele também irá indagar o PC para determinar se há qualquer entrada dele proveniente, por exemplo, acesso baseado em Internet ou informações de configuração.

Vinod: Sim, de fato, configurar o sistema é um estado em si.

Doug: Pessoal, vocês estão enrolando.

Pensemos um pouco mais... Existe uma maneira de colocar esta coisa em um diagrama?

Facilitador: Existe, mas adiemos para logo depois da reunião.

5.5.2 Padrões de análise

Qualquer um que tenha feito engenharia de requisitos em mais do que uns poucos projetos de software começa a perceber a recorrência de certos problemas ao longo de todos os projetos em um campo de aplicação específico.¹⁸ Tais *padrões de análise* [Fow97] sugerem soluções (por exemplo, uma classe, função ou comportamento) no campo de aplicação que pode ser reutilizado na modelagem de muitas aplicações.

Geyer-Schulz e Hahsler [Gey01] sugerem dois benefícios que podem estar associados ao uso de padrões de análise:

Primeiro, os padrões de análise aceleram o desenvolvimento de modelos de análise abstratos que capturam os principais requisitos do problema concreto, fornecendo modelos de análise reutilizáveis com exemplos, bem como uma descrição de vantagens e limitações. Em segundo lugar, os padrões de análise facilitam a transformação do modelo de análise em um modelo de projeto, sugerindo padrões de projeto e soluções confiáveis para problemas comuns.

Os padrões de análise são integrados ao modelo de análise por meio de referência ao nome do padrão. Eles também são armazenados em um repositório de modo que os engenheiros de requisitos podem usar recursos de busca para encontrá-los e aplicá-los. Informações sobre um padrão de análise (e outros tipos de padrões) são apresentadas em um modelo padrão [Gey01]¹⁹ discutido de maneira mais detalhada no Capítulo 12. Exemplos de padrões de análise e uma discussão mais ampla deste tópico são apresentados no Capítulo 7.

5.6 NEGOCIAÇÃO DE REQUISITOS

“Compromisso é a arte de dividir um bolo de tal forma que cada um acredite que ficou com o pedaço maior.”

Ludwig Erhard

WebRef

Um breve artigo sobre negociação para requisitos de software pode ser baixado de www.alexanderegged.com/publications/Software_Requirements_Negotiation_Some_Lessons_Learned.html.

Em um contexto de engenharia de requisitos ideal, as tarefas de início, levantamento e elaboração determinam os requisitos do cliente com detalhes suficientes para prosseguir para atividades de engenharia de software subsequentes. Infelizmente, isso raramente acontece. Na realidade, talvez você tenha de iniciar uma negociação com um ou mais interessados. Na maioria dos casos, solicita-se aos interessados contrabalançar funcionalidade, desempenho e outras características do produto ou sistema, em função do custo e tempo para chegar ao mercado. O intuito da negociação é desenvolver um plano de projeto que atenda às necessidades dos interessados e, ao mesmo tempo, reflita as restrições do mundo real (por exemplo, tempo, pessoal, orçamento) impostas à equipe de software.

As melhores negociações buscam ao máximo um resultado “ganha-ganha”.²⁰ Ou seja, os interessados ganham obtendo um sistema ou produto que satisfaz a maioria de suas necessidades e você (como membro da equipe de software) ganha trabalhando com prazos de entrega e orçamentos reais e atingíveis.

Boehm [Boe98] define um conjunto de atividades de negociação no início de cada iteração do processo de software. Mais do que uma simples atividade de comunicação com o cliente, são definidas as seguintes atividades:

1. Identificação dos principais interessados no sistema ou subsistema.
2. Determinação das “condições de ganho” dos interessados.
3. Negociação das condições de ganho dos interessados para reconciliá-las em um conjunto de condições “ganha-ganha” para todos os envolvidos (inclusive a equipe de software).

O êxito na concretização dessas etapas atinge um resultado em que todos saem ganhando, critério-chave para prosseguir para atividades de engenharia de software subsequentes.

¹⁸ Em alguns casos, essa recorrência de problemas acontece independentemente do campo de aplicação. Por exemplo, as características e funções usadas para resolver problemas de interfaces do usuário são comuns independentemente do campo de aplicação considerado.

¹⁹ Uma grande variedade de modelos de padrões foi proposta na literatura. Caso tenha interesse, consulte [Fow97], [Gam95], [Yac03] e [Bus07] entre as muitas fontes.

²⁰ Foram escritos dezenas de livros sobre habilidades de negociação (por exemplo, [Lew06], [Rai06], [Fis06]). É uma das mais importantes habilidades que você pode aprender. Leia um deles.

INFORMAÇÕES

**A arte da negociação**

Aprender a negociar eficazmente pode ser bem útil para toda a sua vida técnica e pessoal. Vale muito considerar as seguintes diretrizes:

1. *Reconhecer que não se trata de uma competição.* Para ser bem-sucedido, ambas as partes têm de ter a sensação de que ganharam ou atingiram algum objetivo. Ambas terão de ceder.
2. *Planejar uma estratégia.* Decidir o que você quer atingir; o que a outra parte quer atingir e como você irá fazer para que ambas aconteçam.
3. *Ouvir ativamente.* Não trabalhe na formulação de sua resposta enquanto a outra parte estiver falando. Ouça-a. É pro-

vável que você tome conhecimento de algo que irá ajudá-lo a negociar melhor sua posição.

4. *Concentrar-se nos interesses da outra parte.* Não assuma posições rígidas caso queira evitar conflitos.
5. *Não deixar a negociação ir para o lado pessoal.* Concentre-se no problema que precisa ser resolvido.
6. *Ser criativo.* Não tenha medo de inovar caso se encontre em um impasse.
7. *Estar preparado para se comprometer.* Uma vez que se tenha chegado a um acordo, seja objetivo; comprometa-se e vá adiante.

CASASEGURA**O início de uma negociação**

Cena: Sala da Lisa Perez, após a primeira reunião para levantamento de requisitos.

Atores: Doug Miller, gerente de engenharia de software e Lisa Perez, gerente de marketing.

Conversa:

Lisa: Bem, ouvi dizer que a primeira reunião transcorreu muito bem.

Doug: Na verdade, foi assim mesmo. Você mandou bons representantes do seu departamento para a reunião... Eles realmente contribuíram.

Lisa (sorrindo): É, na verdade eles me contaram que se engajaram no processo e que não foi uma "atividade de torrar os miolos".

Doug (rindo): Tomarei cuidado para me despojar do jargão técnico na próxima vez que eu visitar... Veja, Lisa, acho que vamos ter problemas para entregar toda a funcionalidade para o sistema de segurança domiciliar nas datas que sua gerência está propondo. Eu sei, é prematuro, porém já andei fazendo um planejamento preliminar e...

Lisa (franzindo a testa): Temos de ter isso até aquela data, Doug. De que funcionalidade você está falando?

Doug: Presumo que consigamos ter a funcionalidade de segurança domiciliar completa até a data-limite, mas teremos de postergar o acesso via Internet para a segunda versão.

Lisa: Doug, é o acesso via Internet que dá todo o charme ao CasaSegura. Vamos criar toda a campanha de marketing em torno disso. Temos de ter esse acesso!

Doug: Entendo sua situação, realmente. O problema é que, para lhes dar o acesso via Internet, teremos de construir e ter funcionando um site totalmente seguro. Isso demanda tempo e pessoal. Também teremos de agregar um monte de outras funções na primeira versão... Não acredito que possamos fazer isso com os recursos que temos no momento.

Lisa (ainda franzindo a testa): Entendo, mas temos de descobrir uma maneira de ter tudo isso pronto. É crítico para as funções de segurança domiciliar e para outras funções também... Estas últimas poderão esperar até a próxima versão... Concordo com isso.

Lisa e Doug parecem ter chegado a um impasse, mas eles têm de negociar uma solução para o problema. Poderiam os dois "ganhar" neste caso? Fazendo o papel de mediador, o que você sugeriria?

5.7 VALIDAÇÃO DOS REQUISITOS

À medida que cada elemento do modelo de requisitos é criado, é examinado em termos de inconsistência, omissões e ambiguidade. Os requisitos representados pelo modelo são priorizados pelos interessados e agrupados em pacotes de requisitos que serão implementados como incrementos de software. Uma revisão do modelo de requisitos trata as seguintes questões:

- Cada um dos requisitos é consistente com os objetivos globais para o sistema/produto?
- Todos os requisitos foram especificados no nível de abstração apropriado? Ou seja, algum dos requisitos fornece um nível de detalhe técnico inapropriado no atual estágio?

? Ao revisar os requisitos, que perguntas devo fazer?

- O requisito é realmente necessário ou representa um recurso adicional que talvez não seja essencial para o objetivo do sistema?
- Cada um dos requisitos é limitado e sem ambiguidade?
- Cada um dos requisitos possui atribuição? Ou seja, uma fonte (em geral, um indivíduo específico) é indicada para cada requisito?
- Algum dos requisitos conflita com outros requisitos?
- Cada um dos requisitos é atingível no ambiente técnico que irá abrigar o sistema ou produto?
- Cada um dos requisitos pode ser testado, uma vez implementado?
- O modelo de requisitos reflete, de forma apropriada, a informação, função e comportamento do sistema a ser construído?
- O modelo de requisitos foi “dividido” para expor progressivamente informações mais detalhadas sobre o sistema?
- Os padrões de requisitos foram utilizados para simplificar o modelo de requisitos? Todos os padrões foram validados adequadamente? Todos os padrões são consistentes com os requisitos do cliente?

Essas e outras perguntas devem ser feitas e respondidas para garantir que o modelo de requisitos reflita de maneira precisa as necessidades do interessado e forneça uma base sólida para o projeto.

5.8 RESUMO

As tarefas da engenharia de requisitos são conduzidas para estabelecer uma base sólida para o projeto e a construção. A engenharia de requisitos ocorre durante as atividades de comunicação com o cliente e de modelagem que são definidas para o processo genérico de software. São conduzidas sete funções distintas de engenharia de requisitos — concepção, levantamento, elaboração, negociação, especificação, validação e gestão — pelos membros da equipe de software.

Na concepção do projeto, os interessados estabelecem os requisitos básicos do problema, definem restrições de projeto primordiais e tratam as principais características e funções que têm de estar presentes para que o sistema atenda seus objetivos. Essas informações são refinadas e expandidas durante o levantamento — uma atividade de levantamento de requisitos que faz uso de reuniões com a participação de um facilitador, do QFD e do desenvolvimento de cenários de uso.

A elaboração expande ainda mais as necessidades em um modelo — um conjunto de elementos comportamentais, orientados a fluxos e baseados em cenários e classes. O modelo pode fazer referência a padrões de análise, soluções para problemas de análise recorrentes em diferentes aplicações.

À medida que os requisitos são identificados e o modelo de análise é criado, a equipe de software e outros interessados no projeto negociam a prioridade, disponibilidade e custo relativo de cada requisito. O intuito dessa negociação é desenvolver um plano de projeto realista. Além disso, cada requisito e o modelo de análise como um todo é validado em relação às necessidades do cliente para garantir que será construído o sistema correto.

PROBLEMAS E PONTOS A PONDERAR

5.1. Por que um número muito grande de desenvolvedores de software não dedica muita atenção à engenharia de requisitos? Existiria alguma circunstância em que poderíamos deixá-la de lado?

5.2. Foi lhe dada a responsabilidade de extrair os requisitos de um cliente que lhe diz que está muito ocupado para poder atendê-lo. O que você deveria fazer?

- 5.3.** Discuta alguns dos problemas que ocorrem quando os requisitos têm de ser obtidos de três ou quatro clientes diferentes.
- 5.4.** Por que dizemos que o modelo de análise representa uma reprodução de um sistema em determinado momento?
- 5.5.** Suponhamos que você tenha convencido o cliente (você é um excelente vendedor) a concordar com todas as suas exigências como desenvolvedor. Isso o torna um mestre da negociação? Por quê?
- 5.6.** Desenvolva pelo menos três “perguntas livres de contexto” que você faria a um interessado durante a atividade de concepção.
- 5.7.** Desenvolva um “kit” de levantamento de requisitos. O kit deve incluir um conjunto de diretrizes para realizar uma reunião para levantamento de requisitos e materiais que podem ser utilizados para facilitar a criação de listas e quaisquer outros itens que poderiam ajudar na definição dos requisitos.
- 5.8.** Seu professor irá dividir a classe em grupos de quatro a seis alunos. Metade do grupo irá desempenhar o papel do departamento de marketing e a outra fará o papel da engenharia de software. Sua tarefa é definir os requisitos para a função de segurança do *CasaSegura* descrita neste capítulo. Realize uma reunião para levantamento de requisitos usando as diretrizes apresentadas neste capítulo.
- 5.9.** Desenvolva um caso de uso completo para uma das atividades a seguir:
- Fazer um saque em um caixa eletrônico.
 - Usar seu cartão de débito para uma refeição em um restaurante.
 - Comprar ações usando uma conta de corretagem on-line.
 - Procurar livros (sobre um assunto específico) usando uma livreria on-line.
 - Uma atividade especificada pelo seu professor.
- 5.10.** O que representam as “exceções” nos casos de uso?
- 5.11.** Descreva o que é um *padrão de análise* com suas próprias palavras.
- 5.12.** Usando o modelo apresentado na Seção 5.5.2, sugira um ou mais padrões de análise para os seguintes campos de aplicação:
- Software contábil
 - Software de e-mail
 - Navegadores para a Internet
 - Software de processamento de texto
 - Software para criação de sites
 - Um campo de aplicação especificado pelo seu professor
- 5.13.** Qual o significado de *ganha-ganha* no contexto das negociações durante uma atividade de engenharia de requisitos?
- 5.14.** O que você acha que acontece quando uma validação de requisitos revela um erro? Quem será envolvido na correção do erro?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Pelo fato de ser crucial para o sucesso na criação de qualquer sistema complexo baseado em computadores, a engenharia de requisitos é discutida em uma ampla gama de livros. Hood e seus colegas (*Requirements Management*, Springer, 2007) discutem uma série de questões da engenharia de requisitos que abrangem tanto sistemas quanto a engenharia de software. Young (*The Requirements Engineering Handbook*, Artech House Publishers, 2007) apresenta uma discussão aprofundada das tarefas da engenharia de requisitos. Wiegers (*More About Software Requirements*, Microsoft Press, 2006) aborda várias técnicas práticas para o gerenciamento e o levantamento de requisitos. Hull e seus colegas (*Requirements Engineering*,

2. ed., Springer-Verlag, 2004), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) e Sommerville e Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) são apenas alguns exemplos dos muitos livros dedicados ao assunto. Gottesdiener (*Requirements by Collaboration: Workshops for Defining Requirements*, Addison-Wesley, 2002) fornece úteis orientações para aqueles que precisam estabelecer um ambiente colaborativo com seus interessados em termos de levantamento de requisitos.

Lauesen (*Software Requirements: Styles and Techniques*, Addison-Wesley, 2002) traz uma abrangente pesquisa sobre notação e métodos de análise para levantamento de requisitos. Weigers (*Software Requirements*, Microsoft Press, 1999) e Leffingwell e seus colegas (*Managing Software Requirements: A Use Case Approach*, 2. ed., Addison-Wesley, 2003) apresentam um útil conjunto das melhores práticas para levantamento de requisitos e sugerem diretrizes pragmáticas para a maioria dos aspectos do processo da engenharia de requisitos.

Uma visão da engenharia de requisitos baseada em padrões é descrita por Withall (*Software Requirement Patterns*, Microsoft Press, 2007). Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) discute técnicas avançadas para desenvolvimento de requisitos de software. Windle e Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) discutem a engenharia de requisitos no contexto do Processo Unificado e da notação da UML. Alexander e Steven (*Writing Better Requirements*, Addison-Wesley, 2002) fornecem um breve conjunto de diretrizes para redação clara dos requisitos, representando-as como cenários e revisando o resultado final.

A modelagem de casos de uso muitas vezes é o motor para a criação de todos os demais aspectos do modelo de análise. O assunto é debatido exaustivamente por Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007). Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander e Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Leffingwell e seus colegas (*Managing Software Requirements: A Use Case Approach*, 2. ed., Addison-Wesley, 2003) apresentam um proveitoso conjunto das melhores práticas para levantamento de requisitos. Bittner e Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [Coc01], Armour e Miller (*Advanced Use Case Modeling: Software Sistemas*, Addison-Wesley, 2000) e Kulak e seus colegas (*Use Cases: Requirements in Context*, Addison-Wesley, 2000) abordam o levantamento de requisitos com ênfase na modelagem de casos de uso.

Uma ampla gama de fontes de informação sobre análise e engenharia de requisitos se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevantes para a análise e engenharia de requisitos, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

6

MODELAGEM DE REQUISITOS: CENÁRIOS, INFORMAÇÕES E CLASSES DE ANÁLISE

CONCEITOS- -CHAVE

análise de domínio.	153
análise sintática ..	166
associações.....	176
casos de uso.....	157
classes de análise .	166
diagrama de	
atividades	161
diagrama de raios .	162
modelagem baseada	
em cenários	155
modelagem baseada	
em classes	166
modelagem de	
requisitos	154
modelagem CRC.....	171
modelagem e	
dados.....	163
modelos UML ..	161
pacotes de	
análise	178

No nível técnico, a engenharia de software começa com uma série de tarefas de modelagem que levam a especificação dos requisitos e representação do projeto para o software a ser construído. O modelo de requisitos¹ — na verdade, um conjunto de modelos — é a primeira representação técnica de um sistema.

Em um livro seminal sobre métodos de modelagem de requisitos, Tom DeMarco [DeM79] descreve o processo da seguinte maneira:

Revido reconhecidos problemas e falhas anteriores da fase de análise, sugiro que precisamos realizar os seguintes acréscimos ao nosso conjunto de metas. Os produtos da análise devem apresentar grande facilidade em sua manutenção. Isso se aplica particularmente ao Documento-Alvo [especificação de requisitos de software]. Problemas de tamanho devem ser tratados por meio de método efetivo de fracionamento. A especificação escrita como se fossem romances vitorianos está acabada. Elementos gráficos têm de ser usados sempre que possível. Temos de diferenciar as considerações lógicas [essenciais] das físicas [implementação]... No mínimo, precisamos... Algo que nos ajude a subdividir nossos requisitos e documentar essa subdivisão antes da especificação... Alguns meios de acompanhar e avaliar interfaces... Novas ferramentas para descrever lógica e política, algo melhor do que um texto narrativo.

Embora DeMarco tenha escrito sobre os atributos da modelagem de análise há mais de um quarto de século, seus comentários ainda se aplicam a métodos e notação da modelagem de requisitos moderna.

PANORAMA

O que é? A palavra escrita é um maravilhoso veículo para a comunicação, porém, não é necessariamente a melhor maneira de representar os requisitos de um software. A modelagem de requisitos usa uma combinação das formas textual e diagramática para representar os requisitos de maneira relativamente fácil de entender e, mais importante ainda, simples, para fazer a revisão em termos de correção, completude e consistência.

Quem realiza? Um engenheiro de software (às vezes chamado de “analista”) constrói o modelo usando os requisitos extraídos do cliente.

Por que é importante? Para validar os requisitos do software, é preciso examiná-los segundo uma série de pontos de vista. Neste capítulo consideraremos a modelagem de requisitos sob três perspectivas: modelos baseados em cenários, modelos de dados (informações) e modelos baseados em classes. Cada uma deles representa os requisitos em uma “dimensão” diferente, aumentando, portanto, a probabilidade de serem encontrados erros, inconsistências virem à tona e omissões serem reveladas.

Quais são as etapas envolvidas? A modelagem baseada em cenários representa o sistema sob o ponto de vista do usuário. A modelagem de dados representa o espaço de informações e os objetos de dados que o software irá manipular e os relacionamentos entre eles. A modelagem baseada em classes define objetos, atributos e relacionamentos. Assim que os modelos preliminares forem criados, eles são refinados e analisados para avaliar sua clareza, completude e consistência. No Capítulo 7, estendemos as dimensões da modelagem aqui citadas com representações adicionais, fornecendo uma visão mais robusta dos requisitos.

Qual é o artefato? Uma ampla gama de formas textuais e diagramáticas podem ser escolhidas para o modelo de requisitos. Cada uma das representações dá uma visão de um ou mais dos elementos de modelo.

Como garantir que o trabalho foi realizado corretamente? Os artefatos da modelagem de requisitos têm de ser revisados em termos de correção, completude e consistência. Devem refletir as necessidades de todos os interessados e estabelecer uma base a partir da qual o projeto pode ser conduzido.

¹ Em edições anteriores deste livro, usei o termo modelo de análise, em vez de modelo de requisitos. Nesta edição, decidi usar ambos os termos para representar a atividade de modelagem que define vários aspectos do problema a ser resolvido. Análise é a ação que ocorre à medida que requisitos são obtidos.

6.1 ANÁLISE DE REQUISITOS

A análise de requisitos resulta na especificação de características operacionais do software, indica a interface do software com outros elementos do sistema e estabelece restrições que o software deve atender. Permite ainda que você (independentemente de ser chamado de engenheiro de software, analista ou modelador) elabore mais as necessidades básicas estabelecidas durante as tarefas de concepção, levantamento e negociação, que são parte da engenharia de requisitos (Capítulo 5).

A ação da modelagem de requisitos resulta em um ou mais dos seguintes tipos de modelos:

- Modelos baseados em cenários de requisitos do ponto de vista de vários “atores” do sistema.
- Modelos de dados que representam o domínio de informações para o problema.
- Modelos orientados a classes que representam classes orientadas a objetos (atributos e operações) e a maneira por meio da qual as classes colaboram para atender aos requisitos do sistema.
- Modelos orientados a fluxos que representam os elementos funcionais do sistema e como eles transformam os dados à medida que percorrem o sistema.
- Modelos comportamentais que representam como o software se comporta em consequência de “eventos” externos.

Tais modelos dão a um projetista de software informações que podem ser traduzidas em projetos de arquitetura, de interfaces e de componentes. Finalmente, o modelo de requisitos (e a especificação de requisitos de software) fornecem ao desenvolvedor e ao cliente os meios para verificar a qualidade tão logo o software seja construído.

Neste capítulo, nos concentraremos na modelagem baseada em cenários — uma técnica que está se tornando cada vez mais popular na comunidade da engenharia de software; na modelagem de dados — técnica mais especializada que é particularmente apropriada quando uma aplicação tenha de criar ou manipular um espaço de informações complexo; e a modelagem de classes — uma representação de classes orientadas a objetos e as colaborações resultantes que permitem que um sistema funcione. Os modelos orientados a fluxos, os modelos comportamentais, a modelagem baseada em padrões e os modelos WebApp são discutidos no Capítulo 7.

“Qualquer ‘visão’ individual de requisitos é insuficiente para entender ou descrever o comportamento desejado de um sistema complexo.”

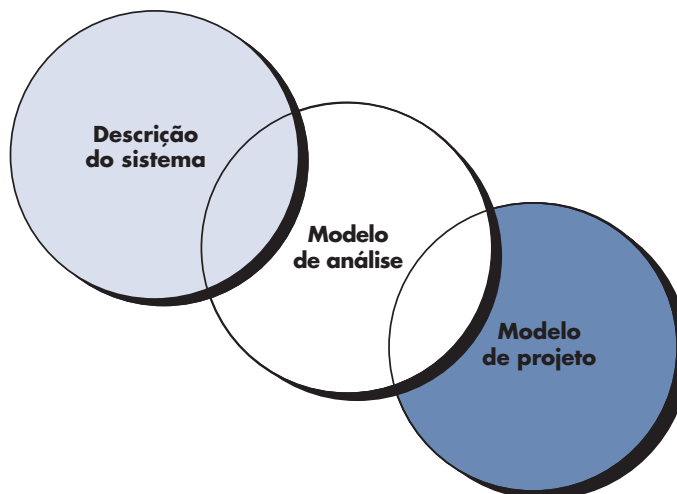
Alan M. Davis

PONTO-CHAVE

O modelo de análise e a especificação de requisitos fornecem um meio para avaliar a qualidade assim que o software for construído.

FIGURA 6.1

O modelo de requisitos como uma ponte entre a descrição do sistema e o modelo de projeto



“Requisito não é sinônimo de arquitetura. Requisito não é sinônimo de projeto nem de interface do usuário. Requisito é sinônimo de necessidade.”

Andrew Hunt e David Thomas

PONTO-CHAVE

O modelo de análise deve descrever o que o cliente quer, deve estabelecer uma base para o projeto, bem como definir uma meta para validação.

6.1.1 Filosofia e objetivos gerais

Na modelagem de requisitos, nosso foco primário é no que e não no como. Que interação com o usuário ocorre em dada circunstância, quais objetos o sistema manipula, que funções o sistema deve executar, quais comportamentos o sistema apresenta, que interfaces são definidas e quais restrições se aplicam?²

Em capítulos anteriores, citamos que a especificação de requisitos completa talvez não seja possível neste estágio. O cliente pode estar inseguro daquilo que é precisamente necessário para certos aspectos do sistema. O desenvolvedor pode estar incerto de que determinada abordagem irá cumprir apropriadamente as funções e o desempenho esperados. Essas realidades atenuam a favor de uma abordagem iterativa para a análise e modelagem de requisitos. O analista deve modelar aquilo que é conhecido e usar o modelo como base para o projeto do incremento de software.³

O modelo de requisitos deve alcançar três objetivos primários: (1) descrever o que o cliente solicita, (2) estabelecer uma base para a criação de um projeto de software e (3) definir um conjunto de requisitos que possa ser validado assim que o software for construído. O modelo de análise preenche a lacuna entre uma descrição sistêmica que descreve o sistema como um todo ou a funcionalidade de negócio que é atingida aplicando-se software, hardware, dados, pessoal e outros elementos de sistema e um projeto de software (Capítulos 8 a 13) que descreve a arquitetura, a interface do usuário e a estrutura em termos de componentes do software. Essa relação é ilustrada na Figura 6.1.

É importante notar que todos os elementos do modelo de requisitos estão diretamente associados a partes do modelo do projeto. Nem sempre é possível estabelecer uma clara divisão das tarefas de análise e de projeto entre essas duas importantes atividades de modelagem. Algum projeto ocorre invariavelmente como parte da análise e alguma análise será realizada durante o projeto.

6.1.2 Regras práticas para a análise

Arlow e Neustadt [Arl02] sugerem uma série de regras práticas que devem ser seguidas ao se criar o modelo de análise:

- *O modelo deve enfatizar as necessidades visíveis no domínio do problema ou do negócio. O nível de abstração deve ser relativamente elevado. “Não se perca nos detalhes” [Arl02] que tentam explicar como o sistema irá funcionar.*
- *Cada elemento do modelo de requisitos deve contribuir para o entendimento geral dos requisitos de software e fornecer uma visão do domínio de informação, função e comportamento do sistema.*
- *Postergue considerações de infraestrutura e outros modelos não funcionais até a fase de projeto. Ou seja, talvez seja preciso um banco de dados, porém as classes necessárias para sua implementação, as funções necessárias para acessar o banco de dados e o comportamento que será apresentado à medida que ele for usado devem ser considerados apenas depois da análise do domínio do problema ter sido completada.*
- *Minimize o acoplamento do sistema. É importante representar os relacionamentos entre as classes e funções. Entretanto, se o nível de “interconexão” for extremamente alto, deve-se esforçar para reduzi-lo.*
- *Certifique-se de que o modelo de requisitos agrega valor a todos os interessados. Cada participante tem um uso próprio para o modelo. Por exemplo, os interessados no negócio devem usar o modelo para validar os requisitos; os projetistas devem usar o modelo como base para o projeto; o pessoal da Garantia da Qualidade (QA) deve usar o modelo para ajudar no planejamento de testes de aceitação.*

? Existem diretrizes básicas que podem nos ajudar enquanto realizamos atividades de análise de requisitos?

“Problemas que valem a pena ser atacados demonstram seu valor contra-atacando.”

Piet Hein

2 Deve-se notar que à medida que os clientes se tornam tecnologicamente mais sofisticados, há uma tendência no sentido da especificação do *como* e também do *quê*. Entretanto, o foco principal deve permanecer no *quê*.
3 Como alternativa, a equipe de software poderia optar por criar um protótipo (Capítulo 2) em uma tentativa de entender melhor as necessidades do sistema.

- Mantenha o modelo o mais simples possível. Não crie diagramas adicionais quando não acrescentam nenhuma informação nova. Não utilize formas de notação complexas, quando uma lista simples já bastaria.

WebRef

Vários recursos úteis para a análise de domínio podem ser encontrados em www.iturils.com/English/SoftwareEngineering/SE_mod5.asp.

6.1.3 Análise de domínio

Na discussão da engenharia de requisitos (Capítulo 5), destaquei que frequentemente ocorre a recorrência de padrões de análise em muitas aplicações em um domínio de aplicação específico. Se esses padrões são definidos e categorizados de maneira que permita seu reconhecimento e sua aplicação para resolver problemas comuns, a criação do modelo de análise deve ser acelerada. Mais importante ainda, a probabilidade de aplicação de padrões de projeto e de componentes de software executáveis cresce dramaticamente. Isso melhora o tempo de colocação do produto no mercado e reduz custos de desenvolvimento.

Mas como os padrões e as classes de análise são inicialmente reconhecidos? Quem os define, os classifica e os prepara para uso em projetos subsequentes? As respostas a essas questões caem na análise de domínio. Firesmith [Fir93] descreve análise de domínio da seguinte maneira:

Análise de domínio de um software é a identificação, a análise e a especificação de requisitos comuns de um campo de aplicação específico, tipicamente para reutilização em vários projetos dentro deste campo de aplicação... [Análise de domínio orientada a objetos é] a identificação, análise e especificação de capacidades comuns reutilizáveis dentro de um campo de aplicação específico, em termos de objetos, classes, componentes e frameworks comuns.

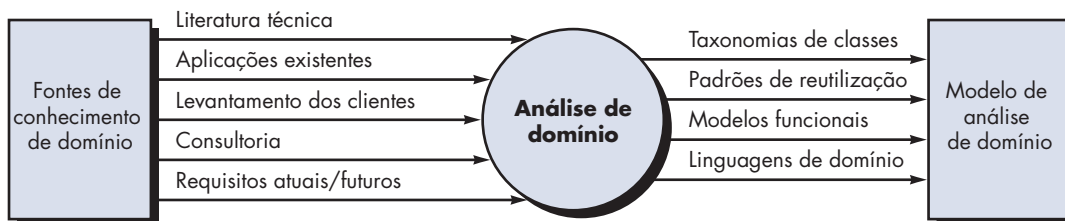
O “domínio de aplicação específico” pode ir da aviônica a sistemas bancários, de videogames multimídia a software embutido em equipamentos médicos. O objeto da análise de domínio é simples: encontrar ou criar aquelas classes de análise e/ou padrões de análise largamente aplicáveis de modo que possam ser reutilizados.⁴

Usando a terminologia introduzida anteriormente neste livro, a análise de domínio pode ser vista como uma atividade universal para o processo de software. Queremos dizer com isso que a análise de domínio é uma atividade contínua da engenharia de software que não está ligada a nenhum projeto de software específico. De certo modo, o papel de um analista de domínio é similar ao papel de um mestre-ferramenteiro em um ambiente de manufatura pesada. O trabalho do mestre-ferramenteiro é projetar e construir ferramentas que possam ser usadas por muitas pessoas que fazem trabalhos semelhantes, mas não necessariamente iguais. O papel da análise de domínio⁵ é descobrir e definir padrões de análise, classes de análise e informações relacionadas que possam ser usados por várias pessoas que trabalham em aplicações similares, mas não necessariamente as mesmas.

A Figura 6.2 [Ara89] ilustra entradas e saídas fundamentais para o processo de análise de domínio. São consultadas fontes de conhecimento de domínio para identificar objetos que possam ser reutilizados no domínio.

FIGURA 6.2

Entrada e saída da análise de domínio



⁴ Uma visão complementar da análise de domínio “envolve a modelagem do domínio de forma que os engenheiros de software e outros interessados possam entender melhor sobre ela... Nem todas as classes de domínio resultam necessariamente no desenvolvimento de classes reutilizáveis...” [Let03a].

⁵ Não parta do pressuposto que pelo fato de um analista de domínio estar envolvido no trabalho, um engenheiro de software não precise entender o domínio de aplicação. Todos os membros da equipe de software precisam ter algum conhecimento do domínio em que o software será inserido.

CASA SEGURA

**Análise de domínio**

Cena: Sala do Doug Miller, após uma reunião com o pessoal de marketing.

Atores: Doug Miller, gerente de engenharia de software e Vinod Raman, membro da equipe de engenharia de software.

Conversa:

Doug: Preciso de você para um projeto especial, Vinod. Vou ter de tirá-lo das reuniões para levantamento de requisitos.

Vinod (franzindo a testa): Que pena. Aquele formato funciona realmente... Eu estava conseguindo extrair algo dessas reuniões. O que acontece?

Doug: Jamie e Ed te substituirão. De qualquer forma, o Marketing insiste que liberemos a capacidade de acesso à Internet com a função de segurança domiciliar na primeira versão do CasaSegura. Estamos com a corda no pescoço nesta... Não temos tempo ou gente suficiente; portanto, temos de resolver ambos os problemas — a interface PC e a interface Web — de uma só vez.

Vinod (parecendo confuso): Não sabia que o plano havia sido estabelecido... Nós ainda nem terminamos de fazer o levantamento de requisitos.

Doug (com um sorriso amarelo): Eu sei, mas os prazos são tão apertados que decidi começar a traçar uma estratégia com o Marketing agora mesmo... De qualquer maneira, revisaremos qualquer plano provisório assim que tivermos todas as informações obtidas nas reuniões de necessidades.

Vinod: OK, o que acontece? O que você quer que eu faça?

Doug: Você sabe o que é “análise de domínio”?

Vinod: Mais ou menos. Primeiramente, devo procurar padrões similares nas aplicações que fazem os mesmos tipos de coisas do que a que estou construindo. Se possível, você “rouba” então os padrões e os reutiliza em seu trabalho.

Doug: Não estou certo da palavra roubar, mas basicamente você entendeu a questão. O que eu gostaria que você fizesse é que começasse a pesquisar interfaces do usuário existentes de sistemas que controlam algo como o CasaSegura. Quero que você proponha um conjunto de padrões e classes de análise que possam ser comuns tanto para a interface baseada em PCs que ficarão instalados nos domicílios quanto a interface baseada em navegadores que pode ser acessada via Internet.

Vinod: Podemos poupar tempo fazendo com que eles sejam os mesmos... Por que não fazemos simplesmente isso?

Doug: Ah... É bom ter pessoas que pensem como você. Esse é o ponto — podemos poupar tempo e esforço se ambas as interfaces forem praticamente idênticas, implementadas com o mesmo código, blá, blá, blá, que o Marketing tanto insiste.

Vinod: Então, você quer o quê? Classes, padrões de análise, padrões de projeto?

Doug: Todos eles. Nada formal neste momento. Quero apenas ter alguma vantagem inicial em nossa análise interna e trabalho de projeto.

Vinod: Pesquisaréi nossa biblioteca de classes e verei o que conseguimos. Também usarei um modelo de padrões que vi em um livro alguns meses atrás.

Doug: Muito bom. Mãos à obra!

“... análise é frustrante, cheia de complexos relacionamentos interpessoais, indefinida e difícil. Resumindo: é fascinante. Uma vez fisgado por ela, os antigos e fáceis prazeres da construção de um sistema jamais serão suficientes para satisfazê-lo novamente.”

Tom DeMarco

6.1.4 Abordagens à modelagem de requisitos

Uma visão da modelagem de requisitos, chamada *análise estruturada*, considera os dados e os processos que transformam os dados em entidades separadas. Os objetos de dados são modelados de maneira que definam seus atributos e relacionamentos. Processos que manipulam objetos de dados são modelados de maneira que mostrem como transformam dados à medida que os objetos de dados trafegam pelo sistema.

Uma segunda abordagem à modelagem de análise, denominada análise orientada a objetos, se concentra na definição de classes e na maneira pela qual colaboram entre si para atender às necessidades dos clientes. A UML e o Processo Unificado (Capítulo 2) são predominantemente orientados a objetos.

Embora o modelo de análise de requisitos proposto neste livro combine ambas as abordagens, as equipes de software em geral optam por uma abordagem e excluem todas as representações da outra. A questão não é qual é a melhor, mas sim, que combinação de representações dará aos interessados o melhor modelo de requisitos de software e a ponte mais efetiva com o projeto de software.

Cada elemento do modelo de requisitos (Figura 6.3) apresenta o problema segundo um ponto de vista. Os elementos baseados em cenários representam como o usuário interage com o sistema e a sequência específica de atividades que ocorre à medida que o software é

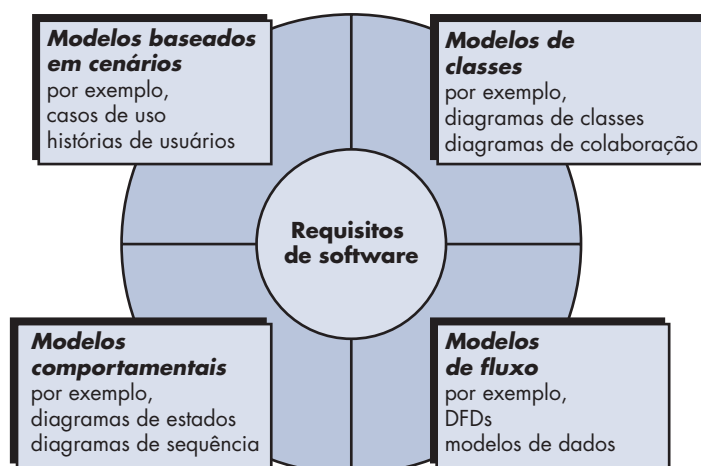
FIGURA 6.3

Elementos do modelo de análise

? Quais pontos de vista diferentes podem ser usados para descrever o modelo de requisitos?

“Por que devemos construir modelos? Por que não construir apenas o sistema? A resposta é que podemos construir modelos para destacar, ou enfatizar, certas características críticas de um sistema e, ao mesmo tempo, diminuir a importância de outros aspectos do sistema.”

Ed Yourdon



utilizado. Os elementos baseados em classes modelam os objetos que o sistema irá manipular, as operações que serão aplicadas aos objetos para efetuar a manipulação, os relacionamentos (alguns hierárquicos) entre os objetos e as colaborações que ocorrem entre as classes definidas. Os elementos comportamentais representam como eventos externos mudam o estado do sistema ou as classes que nele residem. Por fim, elementos orientados a fluxos representam o sistema como uma transformação de informações, indicando como os objetos de dados são transformados à medida que fluem pelas várias funções do sistema.

A modelagem de análise leva à obtenção de cada um desses elementos. Entretanto, o conteúdo específico de cada elemento (os diagramas usados para construir o elemento e o modelo) pode diferir de projeto para projeto. Como já citado várias vezes neste livro, a equipe de software tem de se esforçar para mantê-lo o mais simples possível. Devem ser utilizados apenas aqueles elementos da modelagem que agregam valor ao modelo.

6.2 MODELAGEM BASEADA EM CENÁRIOS

Embora o sucesso de um sistema ou produto baseado em computadores seja medido de muitas formas, a satisfação do usuário encontra-se no topo da lista. Se você entender como os usuários finais (e outros atores) querem interagir com um sistema, sua equipe de software estará mais capacitada a caracterizar, de maneira apropriada, os requisitos e a construir modelos de análise e projeto proveitosos.

Portanto, a modelagem de requisitos com UML⁶ começa com a criação de cenários na forma de casos de uso, diagramas de atividades e diagramas de raias.

6.2.1 Criação de um caso de uso preliminar

Alistair Cockburn caracteriza um caso de uso como um “contrato de comportamento” [Coc01b]. Conforme discutido no Capítulo 5, o “contrato” define a maneira através da qual um ator⁷ usa um sistema baseado em computadores para atingir alguma meta. Em essência, um caso de uso captura as interações que ocorrem entre produtores e consumidores de informação

“[Casos de uso] são apenas uma ferramenta para definir o que existe fora do sistema (atores) e o que deve ser realizado pelo sistema (casos de uso).”

Ivar Jacobson

⁶ A UML será usada como notação para modelagem ao longo deste livro. O Apêndice 1 apresenta um breve tutorial para aqueles que talvez não estejam familiarizados com a notação básica da UML.

⁷ Ator não é uma pessoa específica, mas sim um papel que uma pessoa (ou dispositivo) desempenha em um contexto específico. Um ator “invoca o sistema para que este realize um de seus serviços” [Coc01b].

e o sistema em si. Nesta seção, examinaremos como os casos de uso são desenvolvidos como parte da atividade da modelagem de requisitos.⁸



Em algumas situações, os casos de uso se tornam o mecanismo dominante de engenharia de requisitos. Entretanto, isso não significa que devemos descartar outros métodos de modelagem quando forem apropriados.

No Capítulo 5, citei que um caso de uso descreve um cenário de uso específico em uma linguagem simples sob o ponto de vista de um ator definido. Mas como saber: (1) sobre o que escrever, (2) quanto escrever a seu respeito, (3) com que nível de detalhamento fazer uma descrição e (4) como organizar a descrição? Essas são as questões que devem ser respondidas nas situações em que os casos de uso devam fornecer valor como uma ferramenta da modelagem de requisitos.

Sobre o que escrever? As duas primeiras tarefas da engenharia de requisitos — concepção e levantamento — nos fornecem informações necessárias para começarmos a escrever casos de uso. As reuniões para levantamento de requisitos, o QFD e outros mecanismos de engenharia de requisitos são utilizados para identificar interessados, definir o escopo do problema, especificar as metas operacionais globais, estabelecer as prioridades, descrever todos os requisitos funcionais conhecidos e descrever os itens (objetos) manipulados pelo sistema.

Para começar a desenvolver um conjunto de casos de uso, enumere as funções ou atividades realizadas por um ator específico. Podemos obtê-las de uma lista de funções dos requisitos do

CASA SEGURA



Desenvolvimento de outro cenário preliminar de usuário

Cena: Uma sala de reuniões, durante a segunda reunião para levantamento de requisitos.

Atores: Jamie Lazar, membro da equipe de software; Ed Robins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto e um facilitador.

Conversa:

Facilitador: É hora de começarmos a falar sobre a função de vigilância do CasaSegura. Vamos desenvolver um cenário de usuário para acesso à função de vigilância.

Jamie: Quem desempenha o papel do ator nisto?

Facilitador: Acredito que a Meredith (uma pessoa do Marketing) venha trabalhando nessa funcionalidade. Por que você não desempenha o papel?

Meredith: Você quer fazer da mesma forma que fizemos da última vez, não é mesmo?

Facilitador: Correto... Da mesma forma.

Meredith: Bem, obviamente a razão para a vigilância é permitir ao proprietário do imóvel verificar a casa enquanto ele se encontra fora, gravar e reproduzir imagens de vídeo que são capturadas... Esse tipo de coisa.

Ed: Usaremos compressão para armazenamento de vídeo?

Facilitador: Boa pergunta, Ed, mas vamos postergar essas questões de implementação por enquanto. Meredith?

Meredith: Ok, então basicamente há duas partes para a função de vigilância... A primeira configura o sistema inclusive desenhando uma planta — precisamos de ferramentas para ajudar o proprietário do imóvel a fazer isto — e a segunda parte é a própria função de vigilância real. Como o layout faz parte da atividade de configuração, irei me concentrar na função de vigilância.

Facilitador (sorrindo): Você tirou as palavras de minha boca.

Meredith: Eu quero ter acesso à função de vigilância via PC ou via Internet. Meu sentimento é que o acesso via Internet seria mais utilizado. De qualquer maneira, quero poder exibir visões de câmeras em um PC e controlar o deslocamento e ampliação de imagens de determinada câmera. Especifico a câmera selecionando-a da planta da casa. Quero, de forma seletiva, gravar imagens geradas por câmeras e reproduzi-las. Também quero ser capaz de bloquear o acesso a uma ou mais câmeras com uma senha específica. Também quero a opção de ver pequenas janelas que mostrem visões de todas as câmeras e então poder escolher uma que deseje ampliar.

Jamie: Estas são chamadas de visões em miniatura.

Meredith: OK, então eu quero visões em miniatura de todas as câmeras. Também quero que a interface para a função de vigilância tenha o mesmo aspecto de todas as demais do CasaSegura. Quero que ela seja intuitiva, significando que não vou precisar ler todo o manual para usá-la.

Facilitador: Bom trabalho. Agora, vamos nos aprofundar um pouco mais nesta função...

⁸ Os casos de uso são uma parte importante da modelagem de análise para as interfaces do usuário. A análise de interfaces é discutida em detalhes no Capítulo 11.

sistema, por meio de conversações com interessados ou através de uma avaliação de diagramas de atividades (Seção 6.3.1) desenvolvidas como parte da modelagem de análise.

A função de vigilância domiciliar do CasaSegura (subsistema) discutida no quadro anterior identifica as seguintes funções (uma lista resumida) realizadas pelo ator **proprietário**:

- Selecionar câmera a ser vista.
- Solicitar imagens em miniatura de todas as câmeras.
- Exibir imagens das câmeras em uma janela de um PC.
- Controlar deslocamento e ampliação de uma câmera específica.
- Gravar, de forma seletiva, imagens geradas pelas câmeras.
- Reproduzir as imagens geradas pelas câmeras.
- Acessar a vigilância por câmeras via Internet.

À medida que as conversações com o interessado (que desempenha o papel de proprietário de um imóvel) forem avançando, a equipe de levantamento de requisitos desenvolve casos de uso para cada uma das funções citadas. Em geral, os casos de uso são escritos primeiro de forma narrativa informal. Caso seja necessária maior formalidade, o mesmo caso de uso é reescrito usando um formato estruturado similar àquele proposto no Capítulo 5 e reproduzido mais adiante, nesta seção, na forma de um quadro.

Para fins de ilustração, consideremos a função *acessar a vigilância por câmeras via Internet — exibir visões das câmeras* (**AVC-EVC**). O interessado que faz o papel do ator **proprietário** escreveria a seguinte narrativa:

Caso de uso: Acessar vigilância por câmeras via Internet — exibir visões das câmeras (AVC-EVC)

Ator: proprietário

Se eu estiver em um local distante, posso usar qualquer PC com navegador apropriado para entrar no site Produtos da CasaSegura. Introduzo meu ID de usuário e dois níveis de senhas e, depois de validado, tenho acesso a toda funcionalidade para o meu sistema CasaSegura instalado. Para acessar a visão de câmera específica, seleciono “vigilância” dos botões para as principais funções mostradas. Em seguida, seleciono “escolha uma câmera”, e a planta da casa é mostrada. Depois, seleciono a câmera em que estou interessado. Como alternativa, posso ver, simultaneamente, imagens em miniatura de todas as câmeras selecionando “todas as câmeras” como opção de visualização. Depois de escolher uma câmera, seleciono “visualização”, e uma visualização com um quadro por segundo aparece em uma janela de visualização identificada pelo ID de câmera. Se quiser trocar de câmeras, seleciono “escolha uma câmera”, e a janela de visualização original desaparece e a planta da casa é mostrada novamente. Em seguida, seleciono a câmera em que estou interessado. Surge uma nova janela de visualização.

Uma variação de um caso de uso narrativo apresenta a interação na forma de uma sequência ordenada de ações de usuário. Cada ação é representada como uma sentença declarativa. Voltando à função **AVC-EVC**, poderíamos escrever:

Caso de uso: Acessar vigilância por câmeras via Internet — exibir visões das câmeras (AVC-EVC)

Ator: proprietário

1. O proprietário do imóvel faz o login no site Produtos da CasaSegura.
2. O proprietário do imóvel introduz seu ID de usuário.
3. O proprietário do imóvel introduz duas senhas (cada uma com pelo menos oito caracteres).
4. O sistema mostra os botões de todas as principais funções.
5. O proprietário do imóvel seleciona a “vigilância” por meio dos botões das funções principais.
6. O proprietário do imóvel seleciona “escolher uma câmera”.
7. O sistema mostra a planta da casa.
8. O proprietário do imóvel seleciona um ícone de câmera da planta da casa.

“Os casos de uso podem ser utilizados em vários processos [de software]. Nosso processo favorito é o iterativo e dirigido por riscos.”

**Geri Schneider e
Jason Winters**

9. O proprietário do imóvel seleciona o botão “visualização”.
10. O sistema mostra uma janela de visualização identificada pelo ID de câmera.
11. O sistema mostra imagens de vídeo na janela de visualização a uma velocidade de um quadro por segundo.

É importante notar que essa apresentação sequencial não leva em consideração quaisquer interações alternativas (a narrativa flui de forma mais natural e representa um número pequeno de alternativas). Casos de uso desse tipo são algumas vezes conhecidos como cenários primários [Sch98a].

6.2.2 Refinamento de um caso de uso preliminar

A descrição de interações alternativas é essencial para um completo entendimento da função a ser descrita por um caso de uso. Consequentemente, cada etapa no cenário primário é avaliado fazendo-se as seguintes perguntas [Sch98a]:

 Como examinar sequências de ações alternativas ao desenvolver um caso de uso?

- *O ator pode fazer algo diferente neste ponto?*
- *Existe a possibilidade de o ator encontrar alguma condição de erro neste ponto? Em caso positivo, qual seria ela?*
- *Existe a possibilidade de o ator encontrar algum outro tipo de comportamento neste ponto (por exemplo, comportamento que é acionado por algum evento fora do controle do ator)? Em caso positivo, qual seria ele?*

Respostas a essas perguntas resultam na criação de um conjunto de *cenários secundários* que fazem parte do caso de uso original, mas representam comportamento alternativo. Consideremos, por exemplo, as etapas 6 e 7 do cenário primário apresentado anteriormente:

6. O proprietário do imóvel seleciona “escolher uma câmera”.
7. O sistema mostra a planta da casa.

Poderia o ator assumir outra atitude neste ponto? A resposta é “sim”. Referindo-se à narrativa que flui naturalmente, o ator poderia optar por ver, simultaneamente, imagens em miniatura de todas as câmeras. Portanto, um cenário secundário poderia ser “Visualizar imagens em miniatura para todas as câmeras.”

Existe a possibilidade de o ator encontrar alguma condição de erro neste ponto? Qualquer número de condições de erro pode ocorrer enquanto um sistema baseado em computadores opera. Nesse contexto, consideramos condições de erro apenas aquelas que provavelmente são resultado direto para realizar a ação nas etapas 6 ou 7. Enfatizando, a resposta à pergunta é “sim”. Uma planta com ícones de câmera talvez jamais tenha sido configurada. Portanto, selecionando “escolher uma câmera” resulta em condição de erro: “Não há nenhuma planta configurada para este imóvel”.⁹ Essa condição de erro se torna um cenário secundário.

Existe a possibilidade de o ator encontrar algum outro tipo de comportamento neste ponto? Mais uma vez a resposta é “sim”. Enquanto as etapas 6 e 7 ocorrem, o sistema pode encontrar uma condição de alarme. Isso resultaria no sistema exibir uma notificação de alerta especial (tipo, local, ação do sistema) e dar ao ator uma série de opções relevantes à natureza do alerta. Pelo fato de o cenário secundário poder ocorrer a qualquer momento para praticamente todas as interações, ele não fará parte do caso de uso **AVC-EVC**. Em vez disso, seria desenvolvido um caso de uso distinto — **Condição de alarme encontrada** — e referido de outros casos de uso conforme a necessidade.

Cada uma das situações descritas nos parágrafos anteriores é caracterizada como uma exceção do caso de uso. Uma *exceção* descreve uma situação (seja ela uma condição de falha, seja

⁹ Nesse caso, outro ator, o **administrador do sistema**, teria de configurar a planta da casa, instalar e inicializar (por exemplo, atribuir um ID de equipamento) todas as câmeras e testar cada uma delas para ter certeza de que ele se encontra acessível através do sistema e através da planta da casa.

uma alternativa escolhida pelo ator) que faz com que o sistema exiba um comportamento um tanto diferente.

Cockburn [Coc01b] recomenda o uso de uma sessão de *brainstorming* para obter um conjunto de exceções relativamente completo para cada caso de uso. Além das três perguntas genéricas sugeridas anteriormente nesta seção, as seguintes questões também deveriam ser exploradas:

- *Existem casos em que ocorre alguma “função de validação” durante este caso de uso?* Isso implica que a função de validação é chamada e poderia ocorrer uma condição de erro potencial.
- *Existem casos em que uma função de suporte (ou ator) parará de responder apropriadamente?* Por exemplo, uma ação de usuário aguarda uma resposta, porém a função que deve responder entra em condição de *time-out*.
- *Existe a possibilidade de um fraco desempenho do sistema resultar em ações do usuário inesperadas ou impróprias?* Por exemplo, uma interface baseada na Web responde de forma muito lenta, fazendo com que um usuário selecione um botão de processamento várias vezes seguidas. Essas seleções entram em fila de forma inapropriada e, por fim, geram condição de erro.

A lista de extensões desenvolvidas como consequência das perguntas e respostas deve ser “racionalizada” [Co01b] por meio dos seguintes critérios: deve ser indicada uma exceção no caso de uso se o software for capaz de detectar a condição descrita e, em seguida, tratar a condição assim que esta for detectada. Em alguns casos, uma exceção irá precipitar o desenvolvimento de um outro caso de uso (para tratar da condição percebida).

6.2.3 Criação de um caso de uso formal

Os casos de uso informais apresentados na Seção 6.2.1 são, algumas vezes, suficientes para a modelagem de requisitos. Entretanto, quando um caso de uso envolve uma atividade crítica ou descreve um conjunto complexo de etapas com um número significativo de exceções, uma abordagem mais formal talvez seja mais desejável.

O caso de uso **AVC-EVC** mostrado no quadro segue uma descrição geral típica para casos de uso formais. O *objetivo no contexto* identifica o escopo geral do caso de uso. A *precondição* descreve aquilo que é conhecido como verdadeiro antes de o caso de uso ser iniciado. O *disparador* identifica o evento ou a condição que “faz com que o caso de uso seja iniciado” [Coc01b]. O *cenário* enumera as ações específicas que o ator deve tomar e as respostas apropriadas do sistema. As *exceções* identificam as situações reveladas à medida que o caso de uso preliminar é refinado (Seção 6.2.2). Cabeçalhos adicionais poderão ou não ser acrescentados e são relativamente autoexplicativos.

WebRef

Quando termina a atividade de criação de casos de uso? Para uma proveitosa discussão sobre este tópico, veja ootips.org/usecases/done.html.

Em muitos casos, não há nenhuma necessidade de criar uma representação gráfica de um cenário de uso. Entretanto, a representação diagramática pode facilitar a compreensão, particularmente quando o cenário é complexo. Conforme já citado anteriormente neste livro, a UML oferece recursos de diagramação de casos de uso. A Figura 6.4 representa um diagrama de caso de uso preliminar para o produto *CasaSegura*. Cada caso de uso é representado por uma elipse. Nesta seção foi discutido apenas o caso de uso **AVC-EVC**.

Toda notação de modelagem tem suas limitações, e o caso de uso não é uma exceção. Assim como qualquer outra forma de descrição escrita, a qualidade de um caso de uso depende de seu(s) autor(es). Se a descrição não for clara, o caso de uso pode ser enganoso ou ambíguo. Um caso de uso que se concentra nos requisitos funcionais e comportamentais geralmente é inapropriado para requisitos não funcionais. Para situações em que o modelo de análise deve ter muitos detalhes e precisão (por exemplo, sistemas com segurança crítica), um caso de uso talvez não seja suficiente.

Entretanto, a modelagem baseada em cenários é apropriada para a grande maioria de todas as situações com as quais você irá deparar como engenheiro de software. Se desenvolvido apropriadamente, o caso de uso pode gerar grandes benefícios como ferramenta de modelagem.

CASA SEGURA



Modelo de caso de uso para vigilância

Caso de uso: Acessar a vigilância por câmeras via Internet — exibir visões das câmeras (AVC-EVC)

2, última modificação: 14 de janeiro feita por V. Raman.

Iteração:

Ator primário:

Proprietário do imóvel.

Objetivo no contexto:

Visualizar imagens de câmera espalhadas pela casa de qualquer ponto remoto via Internet.

Precondições:

O sistema deve estar totalmente configurado; devem ser obtidos ID de usuário e senhas apropriadas.

Disparador:

O proprietário do imóvel decide fazer uma inspeção na casa enquanto se encontra fora.

Cenário:

1. O proprietário do imóvel faz o login no site *Produtos da CasaSegura*.
2. O proprietário introduz seu ID de usuário.
3. O proprietário introduz duas senhas (cada uma com pelo menos oito caracteres).
4. O sistema mostra os botões de todas as principais funções.
5. O proprietário seleciona a "vigilância" por meio dos botões das funções principais.
6. O proprietário seleciona "escolher uma câmera".
7. O sistema mostra a planta da casa.
8. O proprietário seleciona um ícone de câmera da planta da casa.
9. O proprietário seleciona o botão "visualização".
10. O sistema mostra uma janela de visualização identificada pelo ID de câmera.
11. O sistema mostra imagens de vídeo na janela de visualização a uma velocidade de um quadro por segundo.

Exceções:

1. O ID ou senhas são incorretos ou não foram reconhecidos — veja o de uso **Validar ID e senhas**.
2. A função de vigilância não está configurada para este sistema — o sistema mostra a mensagem de erro apropriada; veja o caso de uso **Configurar função de vigilância**.
3. O proprietário seleciona "Visualizar as imagens em miniatura para todas as câmeras" — veja o caso de uso **Visualizar as imagens em miniatura para todas as câmeras**.
4. A planta não está disponível ou não foi configurada — exibir a mensagem de erro apropriada e ver o caso de uso **Configurar planta da casa**.
5. É encontrada uma condição de alarme — veja o caso de uso **Condição de alarme encontrada**.

Prioridade:

Prioridade moderada, a ser implementada após as funções básicas.

Quando disponível:

Terceiro incremento.

Frequência de uso:

Frequência moderada.

Canal com o ator:

Via navegador instalado em PC e conexão Internet.

Atores secundários:

Administrador do sistema, câmeras.

Canais com os atores secundários:

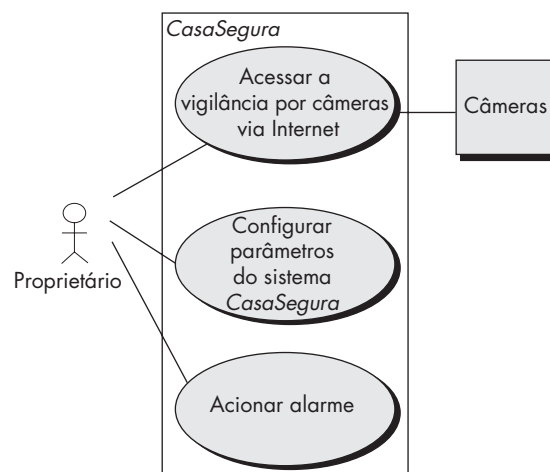
1. Administrador do sistema: sistema baseado em PCs.
2. Câmeras: conectividade sem fio.

Questões abertas:

1. Quais mecanismos protegem o uso não autorizado deste recurso por parte de funcionários da *Produtos da CasaSegura*?
2. A segurança é suficiente? Acessar de forma não autorizada este recurso representaria uma invasão de privacidade importante.
3. A resposta do sistema via Internet seria aceitável dada a largura de banda requerida para visualizações de câmeras?
4. Iremos desenvolver um recurso para fornecer vídeo a uma velocidade de quadros por segundo maior quando as conexões de banda larga estiverem disponíveis?

FIGURA 6.4

Diagrama de caso de uso preliminar para o sistema *CasaSegura*



6.3 MODELOS UML QUE COMPLEMENTAM O CASO DE USO

Existem muitas situações de modelagem de requisitos em que um modelo baseado em texto — mesmo um simples como o caso de uso — talvez não forneça informações de maneira clara e concisa. Em tais casos, pode-se optar por uma ampla gama de modelos gráficos da UML.

6.3.1 Desenvolvimento de um diagrama de atividade

Um diagrama de atividades UML complementa o caso de uso através de uma representação gráfica do fluxo de interação em um cenário específico. Similar ao fluxograma, um diagrama de atividades usa retângulos com cantos arredondados para representar determinada função do sistema, setas para representar o fluxo através do sistema, losangos de decisão para representar uma decisão com ramificação (cada seta saindo do losango é identificada) e as linhas horizontais cheias indicam as atividades paralelas que estão ocorrendo. Um diagrama de atividades para o caso de uso **AVC-EVC** é mostrado na Figura 6.5. Deve-se notar que o diagrama de atividades acrescenta outros detalhes não mencionados (mas implícitos) pelo caso de uso.

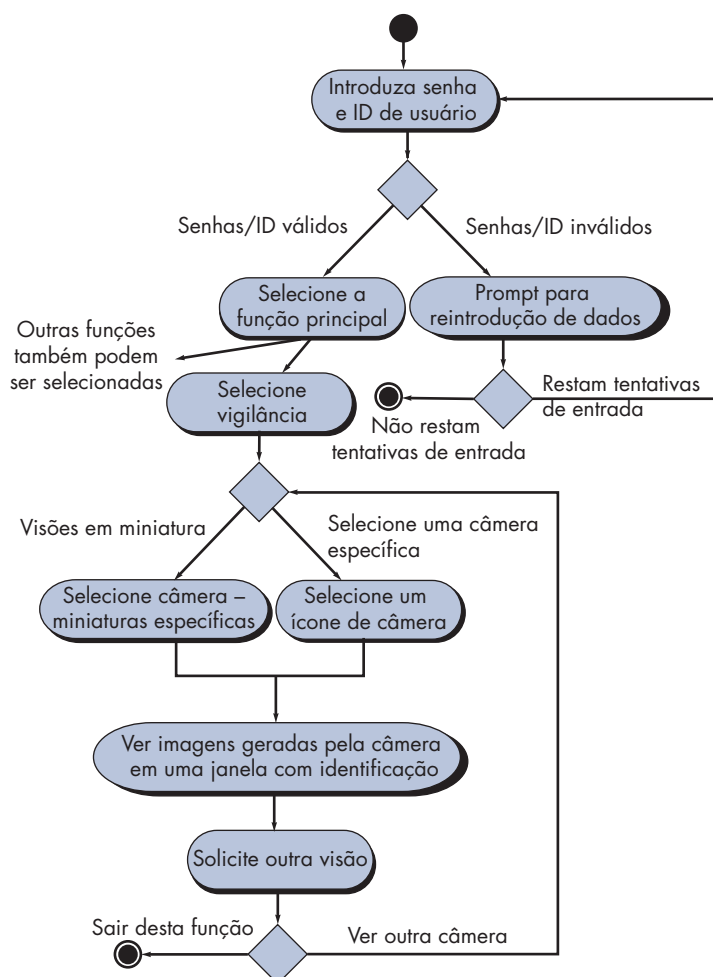
Por exemplo, um usuário poderia tentar introduzir **ID de usuário** e **senha** um número limitado de vezes. Isso é representado pelo losango de decisão abaixo de “Prompt para reintrodução de dados”.

PONTO-CHAVE

Um diagrama de atividades UML representa as ações e decisões que ocorrem enquanto uma dada função é executada.

FIGURA 6.5

Diagrama de atividades para a função **Acessar a vigilância por câmeras via Internet — exibir visões das câmeras**



PONTO-CHAVE

Um diagrama de raias UML representa o fluxo de ações e decisões e indica quais atores realizam cada uma delas.

“Um bom modelo orienta nossas ideias, enquanto um ruim as distorce.”

Brian Marick

6.3.2 Diagramas de raias

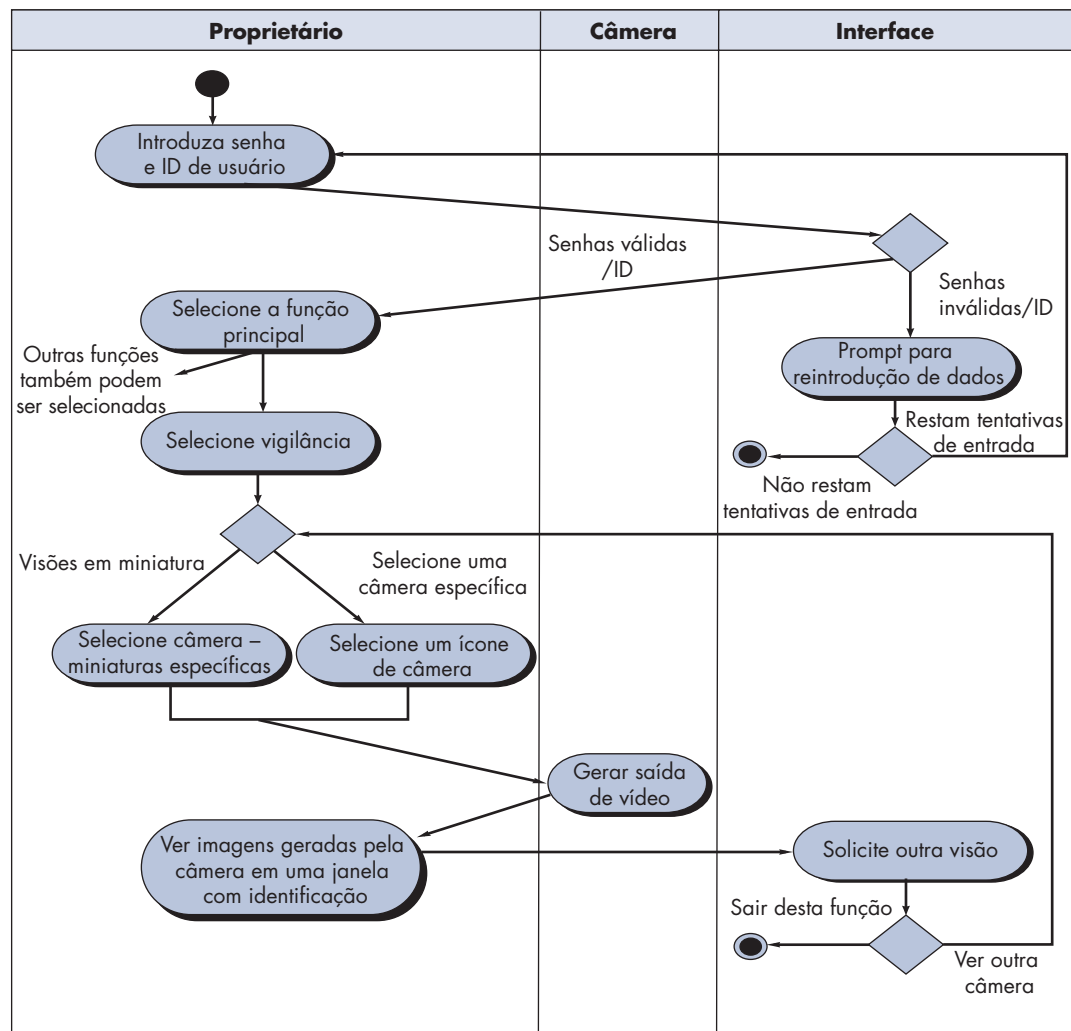
O diagrama de raias UML é uma útil variação do diagrama de atividades e nos permite representar o fluxo de atividades descrito pelo caso de uso e, ao mesmo tempo, indicar qual ator (se existirem vários atores envolvidos em determinado caso de uso) ou classe de análise (discutida posteriormente neste capítulo) tem a responsabilidade pela ação descrita por um retângulo de atividade. As responsabilidades são representadas como segmentos paralelos que dividem o diagrama verticalmente, como as raias de uma piscina.

Três classes de análise — **Proprietário, Câmera e Interface** — possuem responsabilidades diretas ou indiretas no contexto do diagrama de atividades representado na Figura 6.5.

Referindo-se à Figura 6.6, o diagrama de atividades é rearranjado de forma que as atividades associadas a determinada classe de análise caiam na raia para a referida classe. Por exemplo, a classe **Interface** representa a interface do usuário conforme vista pelo proprietário. O diagrama de atividades indica dois *prompts* que são a responsabilidade da interface — “prompt para reintrodução de dados” e “prompt para outra visão”. Esses prompts e as decisões associadas a eles caem na raia **Interface**. Entretanto, as setas saem dessa raia e voltam para a raia do **Proprietário**, onde ocorrem as ações do proprietário do imóvel.

FIGURA 6.6

Diagrama de raias para a função **Acessar a vigilância por câmeras via Internet** — exibir visões das câmeras



Casos de uso, juntamente com os diagramas de atividades e de raias, são orientados a procedimentos. Eles representam a maneira pela qual vários atores chamam funções específicas (ou outras etapas procedurais) para atender aos requisitos do sistema. Porém, uma visão procedural dos requisitos representa apenas uma única dimensão de um sistema. Na Seção 6.4, examinamos o espaço de informações e como os requisitos de dados podem ser representados.

6.4 CONCEITOS DE MODELAGEM DE DADOS

WebRef

Informações úteis sobre a modelagem de dados podem ser encontradas em www.data-model.org.

 Como um objeto de dados se manifesta no contexto de uma aplicação?

PONTO-CHAVE

Objeto de dados é uma representação de qualquer informação composta que é processada pelo software.

PONTO-CHAVE

Os atributos dão nome a um objeto de dados, descrevem suas características e, em alguns casos, fazem referência a um outro objeto.

Se entre os requisitos de software tivermos a necessidade de criar, estender ou interfacear com um banco de dados ou se as estruturas de dados complexas tiverem de ser construídas e manipuladas, a equipe de software poderá optar por criar um *modelo de dados* como parte da modelagem de requisitos. Um analista ou engenheiro de software define todos os objetos de dados processados no sistema, os relacionamentos entre os objetos de dados e outras informações pertinentes aos relacionamentos. O *diagrama entidade-relacionamento* (*entity-relationship diagram*) trata das questões e representa todos os objetos de dados introduzidos, armazenados, transformados e produzidos em uma aplicação.

6.4.1 Objetos de dados

Objeto de dados é uma representação das informações compostas que devem ser compreendidas pelo software. Por *informação composta*, queremos dizer algo que tenha uma série de propriedades ou atributos diferentes. Consequentemente, a largura (um único valor) não seria um objeto de dados válido, mas **dimensões** (incorporando altura, largura e profundidade) poderia ser definido como um objeto.

Um objeto de dados pode ser uma entidade externa (por exemplo, qualquer item que produza ou consuma informações), uma coisa (por exemplo, um relatório ou uma exibição), uma ocorrência (por exemplo, uma ligação telefônica) ou evento (por exemplo, um alerta), um papel (por exemplo, vendedor), uma unidade organizacional (por exemplo, departamento de contabilidade), um local (por exemplo, um armazém) ou uma estrutura (por exemplo, um arquivo). Por exemplo, uma **pessoa** ou um **carro** podem ser vistos como objetos de dados no sentido de que ambos podem ser definidos em termos de um conjunto de atributos. A descrição do objeto de dados incorpora o objeto de dados e todos os seus atributos.

Um objeto de dados encapsula apenas dados — não há nenhuma referência em um objeto de dados a operações que atuam sobre os dados.¹⁰ Consequentemente, o objeto de dados pode ser representado como uma tabela conforme mostrado na Figura 6.7. Os cabeçalhos da tabela refletem os atributos do objeto. Nesse caso, um carro é definido em termos de sua **marca, modelo, número do chassi, tipo de carroceria, cor e proprietário**. O corpo da tabela representa instâncias específicas do objeto de dados. Por exemplo, um Chevy Corvette é uma instância do objeto de dados **carro**.

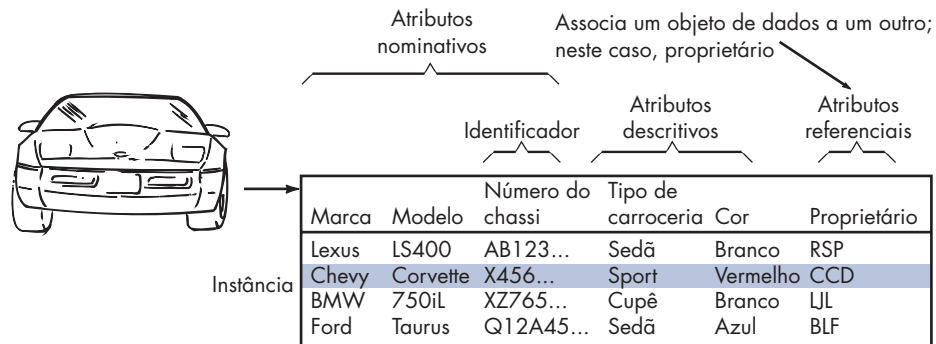
6.4.2 Atributos de dados

Os *atributos de dados* definem as propriedades de um objeto de dados e assumem uma de três características diferentes. Eles podem ser usados para (1) dar um nome a uma instância do objeto de dados, (2) descrever a instância ou (3) fazer referência a uma outra instância em outra tabela. Além disso, um ou mais dos atributos devem ser definidos como identificador — isto é, o atributo identificador se torna uma “chave” quando queremos encontrar uma instância do objeto de dados. Em alguns casos, os valores para o(s) identificador(es) são exclusivos, embora não seja uma exigência. Fazendo referência ao objeto de dados **carro**, um identificador razoável poderia ser o **número do chassi**.

¹⁰ Essa distinção separa o objeto de dados da classe ou objeto definido como parte da abordagem orientada a objetos (Apêndice 2).

FIGURA 6.7

Representação tabular de objetos de dados



WebRef

Um conceito chamado “normalização” é importante para aqueles que pretendem realizar uma modelagem de dados completa. Uma útil introdução pode ser encontrada em www.datamodel.org.

O conjunto de atributos apropriado para um objeto de dados é determinado por meio do entendimento do contexto do problema. Os atributos para **carro** poderiam servir bem para uma aplicação a ser usada por um departamento de veículos a motor, porém tais atributos seriam de pouca valia para uma empresa do setor automobilístico que precisasse fabricar software de controle. Neste último caso, os atributos para **carro** também poderiam incluir o **número do chassi**, o **tipo de carroceria** e a **cor**, porém, muitos outros atributos (por exemplo, **código interno**, **tipo de acionamento do motor**, **indicador do conjunto de tapeçaria**, **tipo de transmissão**) teriam de ser acrescidos para tornar **carro** um objeto com significado no contexto de controle de manufatura.

INFORMAÇÕES



Objetos de dados e classes orientadas a objetos — são eles a mesma coisa?

Uma pergunta que comumente surge quando o tema é objetos de dados: Um objeto de dados é a mesma coisa que uma classe orientada a objetos¹¹? A resposta é “não”.

O objeto de dados define um dado composto; ou seja, ele incorpora um conjunto de dados individuais (atributos) e dá um nome ao conjunto de itens (o nome do objeto de dados).

Uma classe orientada a objetos encapsula atributos de dados, mas também incorpora as operações (métodos) que manipulam os dados decorrentes desses atributos. Além disso, a definição de classes implica uma ampla infraestrutura que faz parte da abordagem de engenharia de software orientada a objetos. As classes se comunicam entre si via mensagens, podem ser organizadas em hierarquias e fornecem características de herança para objetos que são uma instância de uma classe.

6.4.3 Relacionamentos

Os objetos de dados são interligados de várias formas. Consideremos os dois objetos de dados, **pessoa** e **carro**. Estes podem ser representados usando-se a notação simples ilustrada na Figura 6.8a. É estabelecida uma conexão entre **pessoa** e **carro**, pois os dois objetos estão interrelacionados. Mas quais são os relacionamentos? Para determinarmos a resposta, devemos entender o papel das pessoas (proprietários, neste caso) e dos carros no contexto do software a ser construído. Podemos estabelecer um conjunto de pares objeto/relacionamento que definem os relacionamentos relevantes. Por exemplo,

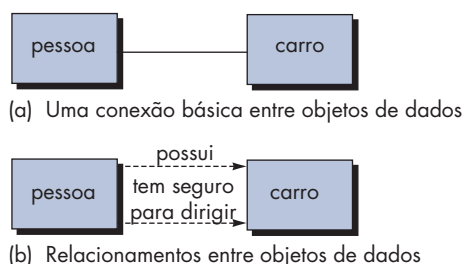
- Uma pessoa *possui* um carro.
- Uma pessoa *tem seguro para dirigir* um carro.

PONTO-CHAVE

Os relacionamentos indicam a maneira por meio da qual os objetos de dados são interligados entre si.

¹¹ Os leitores que não estiverem familiarizados com os conceitos e a terminologia da orientação a objetos devem consultar o breve tutorial apresentado no Apêndice 2.

FIGURA 6.8

Relacionamentos entre objetos de dados

Os relacionamentos *possui* e *tem seguro para dirigir* definem as conexões relevantes entre **pessoa** e **carro**. A Figura 6.8b ilustra graficamente esses pares objeto-relacionamento. As setas indicadas na Figura 6.8b dão uma importante informação sobre a direção do relacionamento e, normalmente, reduzem a ambiguidade ou interpretações incorretas.

**Diagramas entidade-relacionamento**

O par objeto-relacionamento é a pedra angular do modelo de dados. Os pares podem ser representados graficamente por meio do diagrama entidade-relacionamento (DER).¹² O DER foi originalmente proposto por Peter Chen [Che77] para o projeto de sistemas de bancos de dados relacionais e foi estendido por outros. Um conjunto de componentes primordiais é identificado para o DER: objetos de dados, atributos, relacionamentos e indicadores de vários tipos. O principal objetivo do DER é representar objetos de dados e seus relacionamentos.

Já foi feita uma introdução rudimentar sobre a notação do DER. Os objetos de dados são representados por um retângulo identificado. Os relacionamentos são indicados com uma linha rotulada interligando objetos. Em algumas variantes do DER, a linha de conexão contém um losango, rotulado com um relacionamento. As conexões entre objetos de dados e relacionamentos são estabelecidas usando-se uma variedade de símbolos especiais que indicam a cardinalidade e a modalidade.¹³ Se desejar maiores informações sobre modelagem de dados e o diagrama entidade-relacionamento, consulte [Hob06] ou [Sim05].

INFORMAÇÕES**Modelagem de dados**

Objetivo: As ferramentas de modelagem de dados dão a um engenheiro de software a capacidade de representar objetos de dados, suas características e seus relacionamentos. Usado primariamente para aplicações de bancos de dados de grande porte e outros projetos de sistemas de informação, as ferramentas para modelagem de dados fornecem um meio automatizado para criar diagramas entidade-relacionamento completos, dicionários de objetos de dados e modelos relacionados.

Mecânica: As ferramentas nesta categoria permitem ao usuário descrever objetos de dados e seus relacionamentos. Em alguns casos, as ferramentas usam a notação DER. Em outros, as ferramentas modelam relações por meio de algum outro mecanismo. As ferramentas nesta categoria são em geral usadas como parte do projeto de um banco de dados e permitem a criação de um modelo de banco de dados através da geração

FERRAMENTAS DO SOFTWARE

de um esquema de banco de dados para sistemas de gerenciamento de bancos de dados comuns (SGBD).

Ferramentas representativas¹⁴:

AllFusion ERWin, desenvolvida pela Computer Associates (www.3.ca.com), auxilia no projeto de objetos de dados, na estrutura apropriada e elementos-chave para bancos de dados.

ER/Studio, desenvolvida pela Embarcadero Software (www.embarcadero.com), oferece suporte à modelagem entidade-relacionamento.

Oracle Designer, desenvolvida pela Oracle Systems (www.oracle.com), “modela processos de negócio, entidades de dados e relacionamentos [que] são transformados em projetos a partir dos quais são gerados aplicações e bancos de dados completos”.

Visible Analyst, desenvolvida pela Visible Systems (www.visible.com), oferece suporte a uma série de funções de modelagem de análise inclusive à modelagem de dados.

12 Embora o DER ainda seja utilizado em algumas aplicações de projeto de bancos de dados, a notação UML (Apêndice 1) agora pode ser usada para projeto de dados.

13 *Cardinalidade* de um par objeto-relacionamento especifica “o número de ocorrências de um [objeto] que pode ser relacionado ao número de ocorrências de um outro [objeto]” [Til93]. A *modalidade* de um relacionamento é 0 se não houver uma necessidade explícita para a ocorrência do relacionamento ou o relacionamento for opcional. A modalidade é 1 se a ocorrência do relacionamento for compulsória.

14 As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra de ferramentas nesta categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas por seus respectivos desenvolvedores.

6.5 MODELAGEM BASEADA EM CLASSES

A modelagem baseado em classes representa os objetos que o sistema irá manipular, as operações (também denominadas métodos ou serviços) que serão aplicadas aos objetos para efetuar a manipulação, os relacionamentos (alguns hierárquicos) entre os objetos e as colaborações que ocorrem entre as classes definidas. Os elementos de um modelo baseado em classes são: classes e objetos, atributos, operações, modelos CRC (classe-responsabilidade-colaborador), diagramas de colaboração e pacotes. As seções a seguir apresentam uma série de diretrizes informais que auxiliarão na identificação e na representação desses elementos.

6.5.1 Identificação de classes de análise

Se você observar em torno de uma sala, existe um conjunto de objetos físicos que podem ser facilmente identificados, classificados e definidos (em termos de atributos e operações). Porém, quando se “observa em torno” do espaço de problema de uma aplicação de software, as classes (e objetos) podem ser mais difíceis de compreender.

Podemos começar a identificar classes examinando os cenários de uso desenvolvidos como parte do modelo de requisitos e realizando uma “análise sintática” [Abb83] dos casos de uso desenvolvidos para o sistema a ser construído. As classes são determinadas sublinhando-se cada substantivo ou frase contendo substantivos e introduzindo-a em uma tabela simples. Podem ser observados sinônimos. Se for preciso que a classe (substantivo) implemente uma solução, então ela faz parte do espaço de soluções; caso contrário, se for necessária uma classe apenas para descrever uma solução, ela faz parte do espaço de problemas.

Mas o que devemos procurar uma vez que todos os substantivos tenham sido isolados? As *classes de análise* se manifestam em uma das seguintes maneiras:

- *Entidades externas* (por exemplo, outros sistemas, dispositivos, pessoas) que produzem ou consomem informações a ser usadas por um sistema baseado em computadores.
- *Coisas* (por exemplo, relatórios, exibições, letras, sinais) que fazem parte do domínio de informações para o problema.
- *Ocorrências ou eventos* (por exemplo, uma transferência de propriedades ou a finalização de uma série de movimentos de robô) que ocorrem no contexto da operação do sistema.
- *Papéis* (por exemplo, gerente, engenheiro, vendedor) desempenhados pelas pessoas que interagem com o sistema.
- *Unidades organizacionais* (por exemplo, divisão, grupo, equipe) que são relevantes para uma aplicação.
- *Locais* (por exemplo, chão de fábrica ou área de carga) que estabelecem o contexto do problema e a função global do sistema.
- *Estruturas* (por exemplo, sensores, veículos de quatro rodas ou computadores) que definem uma classe de objetos ou classes de objetos relacionadas.

Essa categorização é apenas uma das muitas que foram propostas na literatura.¹⁵ Por exemplo, Budd [Bud96] sugere uma taxonomia de classes que inclui *produtores* (fontes) e *consumidores* (reservatórios) de dados, *gerenciadores de dados*, *classes de visualização ou de observação* e *classes auxiliares*.

Também é importante notar o que as classes ou objetos não devem ser. Em geral, uma classe jamais deve ter um “nome procedural imperativo” [Cas89]. Por exemplo, se os desenvolvedores de software de um sistema de imagens para aplicação em medicina definiram um objeto com o nome **InverterImagem** ou até mesmo **InversãoDeImagem**, eles estariam cometendo um erro sutil. A **Imagem** obtida do software poderia, obviamente, ser uma classe (é algo que faz parte do domínio de informações). A inversão da imagem é uma operação aplicada ao objeto.

“O problema realmente difícil é descobrir, logo no início, quais são os objetos [classes] corretos.”

Carl Argila

 De que maneira as classes de análise se manifestam como elementos do espaço de soluções?

15 Outra classificação importante, definindo classes de entidades, de contorno e de controle, é discutida na Seção 6.5.4.

É provável que a inversão seja definida como uma operação para o objeto **Imagem**, mas não seja definida como uma classe separada com a conotação “inversão de imagem”. Como afirma Cashman [Cas89]: “o intuito da orientação a objetos é encapsular, mas, ainda, manter separados os dados e as operações sobre os dados”.

Para ilustrarmos como as classes de análise poderiam ser definidas durante os estágios iniciais da modelagem, consideremos uma análise sintática (os substantivos são sublinhados, os verbos colocados em *itálico*) para uma narrativa¹⁶ de processamento da *função de segurança domiciliar do CasaSegura*.



A análise sintática não é infalível, mas um excelente salto inicial, caso você esteja em dificuldades para definir objetos de dados e as transformações que operam sobre eles.

A função de segurança domiciliar do CasaSegura *permite* que o proprietário do imóvel *configure* o sistema de segurança quando ele é *instalado*, *monitore* todos os sensores conectados ao sistema de segurança e *interaja* com o proprietário do imóvel através da Internet, de um PC ou de um painel de controle.

Durante a instalação, o PC do CasaSegura é usado para *programar* e *configurar* o sistema. É atribuído um número e um tipo a cada sensor, é programada uma senha-mestra para *armar* e *desarmar* o sistema e são *introduzidos* número(s) de telefone para *discar* quando um evento de sensor ocorre.

Quando um evento de sensor é *reconhecido*, o software *aciona* um alarme audível agregado ao sistema. Após um tempo de retardo que é *especificado* pelo proprietário do imóvel durante as atividades de configuração do sistema, o software *disca* um número de telefone de um serviço de monitoramento, *fornece* informações sobre o local, *relatando* a natureza do evento que foi detectado. O número de telefone será *discado novamente* a cada 20 segundos até que seja *completada* a ligação.

O proprietário do imóvel *recebe* informações de segurança através de um painel de controle, do PC ou de um navegador, coletivamente denominados interface. A interface *mostra* mensagens ao operador, bem como informações sobre o estado do sistema no painel de controle, no PC ou na janela do navegador. A interação com o proprietário do imóvel acontece da seguinte forma...

Extraindo os substantivos, podemos propor uma série de possíveis classes:

Classe potencial	Classificação geral
proprietário do imóvel	papel ou entidade externa
sensor	entidade externa
painel de controle	entidade externa
instalação	ocorrência
sistema (também conhecido como sistema de segurança)	coisa
número, tipo,	não objetos, atributos do sensor
senha-mestra	coisa
número de telefone	coisa
evento de sensor	ocorrência
alarme audível	entidade externa
serviço de monitoramento	unidade organizacional ou entidade externa

Prosseguiríamos com a lista até que todos os substantivos contidos na narrativa de processamento tivessem sido considerados. Observe que chamamos cada entrada da lista de possível objeto. Temos de considerar cada um deles até que uma decisão final seja feita.

¹⁶ Uma narrativa de processamento é similar ao caso de uso em termos de estilo, mas ligeiramente diferente em seu propósito. A narrativa de processamento fornece uma descrição geral da função a ser desenvolvida. Ela não é um cenário escrito sob o ponto de vista de um ator. É importante observar, entretanto, que uma análise sintática também pode ser usada para todos os casos de uso desenvolvidos como parte do levantamento de requisitos (inferência).

? Como podemos determinar se uma classe potencial deve, de fato, se tornar uma classe de análise?

Coad e Yourdon [Coa91] sugerem seis características de seleção que deveriam ser usadas à medida que se considera cada classe potencial para inclusão no modelo de análise:

1. *Informações retidas.* A classe potencial será útil durante a análise apenas se as informações sobre ela tiverem de ser lembradas para que o sistema possa funcionar.
2. *Serviços necessários.* A classe potencial deve ter um conjunto de operações identificáveis capazes de modificar, de alguma forma, o valor de seus atributos.
3. *Atributos múltiplos.* Durante a análise de requisitos, o foco deve ser nas informações “importantes”; uma classe com um único atributo poderia, na verdade, ser útil durante o projeto, porém, provavelmente seria mais bem representada na forma de atributo de outra classe durante a atividade de análise.
4. *Atributos comuns.* Um conjunto de atributos pode ser definido para a classe potencial e esses atributos se aplicam a todas as instâncias da classe.
5. *Operações comuns.* Um conjunto de operações pode ser definido para a classe potencial e tais operações se aplicam a todas as instâncias da classe.
6. *Requisitos essenciais.* Entidades externas que aparecem no espaço do problema e produzem ou consomem informações essenciais à operação de qualquer solução para o sistema quase sempre serão definidas como classes no modelo de requisitos.

“As classes batalham, algumas delas triunfam, outras são eliminadas.”

Mao Zedong

Para ser considerada uma classe legítima para inclusão no modelo de requisitos, um objeto potencial deve satisfazer todas (ou quase todas) essas características. A decisão para inclusão de classes potenciais no modelo de análise é um tanto subjetiva, e uma avaliação posterior poderia fazer com que um objeto fosse descartado ou reintegrado. Entretanto, a primeira etapa da modelagem baseada em classes é definir as classes e decisões (mesmo aquelas subjetivas). Com isso em mente, devemos aplicar as características de seleção da lista de classes potenciais da *CasaSegura*:

Classe potencial	Número característico que se aplica
proprietário do imóvel	rejeitado: 1, 2 falham, embora 6 se aplique
sensor	aceito: todos se aplicam
painel de controle	aceito: todos se aplicam
instalação	rejeitado
sistema (também conhecido como sistema de segurança)	aceito: todos se aplicam
número, tipo	rejeitado: 3 falha, atributos de sensor
senha-mestra	rejeitado: 3 falha
número de telefone	rejeitado: 3 falha
evento de sensor	aceito: todos se aplicam
alarme audível	aceito: 2, 3, 4, 5, 6 se aplicam
serviço de monitoramento	rejeitado: 1, 2 falham, embora 6 se aplique

Deve-se notar o seguinte: (1) a lista anterior não é definitiva, outras classes talvez tenham de ser acrescentadas para completar o modelo; (2) algumas das classes potenciais rejeitadas se tornarão atributos para as que foram aceitas (por exemplo, número e tipo são atributos de **Sensor** e senha-mestre e número de telefone talvez se tornem atributos de **Sistema**); (3) enunciados do problema diferentes talvez provoquem decisões “aceito ou rejeitado” diferentes (por exemplo, se cada proprietário de um imóvel tivesse uma senha individual ou fosse identificado pelo seu padrão de voz, a classe **Proprietário** satisfaria as características 1 e 2 e teriam sido aceitas).

PONTO-CHAVE

Atributos são o conjunto de objetos de dados que definem completamente a classe no contexto do problema.

6.5.2 Especificação de atributos

Os *atributos* descrevem uma classe selecionada para ser incluída no modelo de requisitos. Em essência, são os atributos que definem uma classe — que esclarecem o que a classe representa no contexto do espaço de problemas. Por exemplo, se fôssemos construir um sistema que registrasse estatísticas dos jogadores do campeonato de beisebol profissional, os atributos da classe **Jogador** seriam bem diferentes dos atributos da mesma classe quando usados no contexto do sistema de aposentadoria dos jogadores profissionais de beisebol. No primeiro, atributos como **nome**, **posição**, **média de rebatidas**, **porcentagem de voltas completas em torno do campo**, **número de anos jogados** e **número de jogos** podem ser relevantes. No outro caso, alguns desses atributos seriam relevantes, mas outros substituídos (ou aumentados) por atributos como **salário médio**, **crédito faltante para aposentadoria plena**, **opções escolhidas para o plano de aposentadoria**, **endereço para correspondência** e outros similares.

Para criarmos um conjunto de atributos que fazem sentido para uma classe de análise, devemos estudar cada caso de uso e escolher aquelas “coisas” que “pertencem” de forma razoável àquela classe. Além disso, a pergunta a seguir deve ser respondida para cada uma das classes: “Quais dados (compostos e/ou elementares) definem completamente esta classe no contexto do problema em mãos?”.

A título de ilustração, consideremos a classe **Sistema** definida para o *CasaSegura*. O proprietário de um imóvel pode configurar a função de segurança para que esta reflita informações sobre os sensores, tempo de resposta do alarme, ativação/desativação, de identificação e assim por diante. Podemos representar estes dados compostos da seguinte maneira:

informações de identificação = ID do sistema + número de telefone de verificação + estado do sistema

informações sobre a resposta do alarme = tempo de retardo + número de telefone

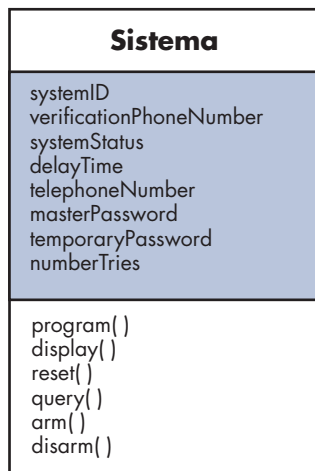
informações de ativação/desativação = senha-mestre + número de tentativas permitidas + senha temporária

Cada um dos dados à direita do sinal de igual poderia ser definido de forma mais extensiva a um nível elementar, porém, para nossos propósitos, eles constituem uma lista de atributos razoável para a classe **Sistema** (a parte sombreada da Figura 6.9).

Os sensores fazem parte do sistema global *CasaSegura* e ainda não estão listados como dados ou atributos na Figura 6.9. **Sensor** já foi definido como uma classe, e múltiplos objetos **Sensor** serão associados à classe **Sistema**. Em geral, evitamos definir um item como um atributo, caso mais de um dos itens deva ser associado à classe.

FIGURA 6.9

Diagrama de classes para a classe Sistema





Ao definir operações para uma classe de análise, concentre-se no comportamento orientado ao problema em vez de nos comportamentos necessários para a implementação.

6.5.3 Definição das operações

As operações definem o comportamento de um objeto. Embora existam muitos tipos diferentes de operações, em geral podem ser divididas em quatro grandes categorias: (1) operações que manipulam dados de alguma forma (por exemplo, adição, eliminação, reformatação, seleção), (2) operações que realizam um cálculo, (3) operações que pesquisam o estado de um objeto e (4) operações que monitoram um objeto em termos de ocorrências de um evento de controle. Tais funções são realizadas operando-se sobre atributos e/ou associações (Seção 6.5.5). Consequentemente, uma operação deve ter “conhecimento” da natureza dos atributos e associações das classes.

Como primeira iteração na obtenção de um conjunto de operações para uma classe de análise, podemos estudar novamente uma narrativa de processamento (ou caso de uso) e escolher aquelas operações que pertencem de forma razoável à classe. Para tanto, a análise sintática é mais uma vez estudada e os verbos são isolados. Alguns dos verbos serão as operações legítimas e podem ser facilmente associadas a uma classe específica. Por exemplo, da narrativa de processamento do *CasaSegura* apresentada anteriormente neste capítulo, veremos que “é atribuído um número e tipo aos sensores” ou “uma senha-mestre é programada para armar e desarmar o sistema”. Essas frases indicam uma série de coisas:

- Que uma operação *assign()* é relevante para a classe **Sensor**.
- Que uma operação *program()* será aplicada à classe **Sistema**.
- Que *arm()* e *disarm()* são operações que se aplicam à classe **Sistema**.

CASA SEGURA



Modelos de classes

Cena: Sala do Ed, quando se inicia o modelamento de requisitos.

Atores: Jamie, Vinod e Ed — todos os membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

[Ed vem trabalhando na extração de classes do modelo de casos de uso para o AVC-EVC (apresentado em um quadro anterior deste capítulo) e está mostrando as classes que ele extraiu a seus colegas.]

Ed: Então, quando o proprietário de um imóvel quiser escolher uma câmera, ele terá de selecioná-la na planta da casa. Defini uma classe **Planta**. Eis o diagrama.

(Eles observam a Figura 6.10.)

Jamie: Portanto, **Planta** é um objeto que é colocado com paredes, portas, janelas e câmeras. É isso que estas linhas com identificação significam, certo?

Ed: Isso mesmo, elas são chamadas “associações”. Uma classe está associada a outra de acordo com as associações que eu mostrei. [As associações são discutidas na Seção 6.5.5.]

Vinod: Portanto, a planta real é feita de paredes e contém câmeras e sensores colocados nessas paredes. Como a planta da casa sabe onde colocar esses objetos?

Ed: Ela não sabe, mas as outras classes sim. Veja os atributos em, digamos, **TrechoParede**, que é usada para construir uma

parede. O trecho de parede tem coordenadas de início e fim e a operação *draw()* faz o resto.

Jamie: E o mesmo acontece com as janelas e portas. Parece-me que câmera possui alguns atributos extras.

Ed: Exatamente, preciso deles para fornecer informações sobre deslocamento e ampliação de imagens.

Vinod: Tenho uma pergunta. Por que a câmera possui um ID, mas os demais não? Percebi que você tem um atributo chamado **próximaParede**. Como **TrechoParede** saberá qual será a parede seguinte?

Ed: Boa pergunta, mas como dizem por aí, essa é uma decisão de projeto e, portanto, a postergarei até...

Jamie: Dá um tempo... Aposto que você já bolou isso.

Ed (sorrindo timidamente): É verdade, vou usar uma estrutura de listas que irei modelar quando chegarmos ao projeto. Se ficarmos muito presos à questão de separar análise e projeto, o nível de detalhe que tenho exatamente aqui poderia ser duvidoso.

Jamie: Para mim parece muito bom, mas tenho algumas outras perguntas.

(Jamie faz perguntas que resultam em pequenas modificações.)

Vinod: Você tem cartões CRC para cada um dos objetos? Em caso positivo, deveríamos simular seus papéis, apenas para ter certeza de que nada tenha escapado.

Ed: Não estou bem certo de como fazê-lo.

Vinod: Não é difícil e realmente vale a pena. Vou lhes mostrar.

Após uma investigação mais apurada, é provável que a operação *program()* seja dividida em uma série de suboperações mais específicas exigidas para configurar o sistema. Por exemplo, *program()* implica a especificação de números de telefone, a configuração de características do sistema (por exemplo, criar a tabela de sensores, introduzir características do alarme) e a introdução de senha(s). Mas, por enquanto, especificaremos *program()* como uma única operação.

Além da análise sintática, pode-se ter uma melhor visão sobre outras operações considerando-se a comunicação que ocorre entre os objetos. Os objetos se comunicam passando mensagens entre si. Antes de prosseguirmos com a especificação de operações, exploraremos essa questão em mais detalhes.

6.5.4 Modelagem CRC (Classe-Responsabilidade-Colaborador)

A modelagem CRC (*classe-responsabilidade-colaborador*) [Wir90] fornece uma maneira simples para identificar e organizar as classes que são relevantes para os requisitos do sistema ou produto. Ambler [Amb95] descreve a modelagem CRC da seguinte maneira:

Um modelo CRC é, na verdade, um conjunto de fichas-padrão que representam classes. Os cartões são divididos em três seções. Ao longo da parte superior do cartão escrevemos o nome da classe. No corpo do cartão enumeramos as responsabilidades da classe do lado esquerdo e os colaboradores do lado direito.

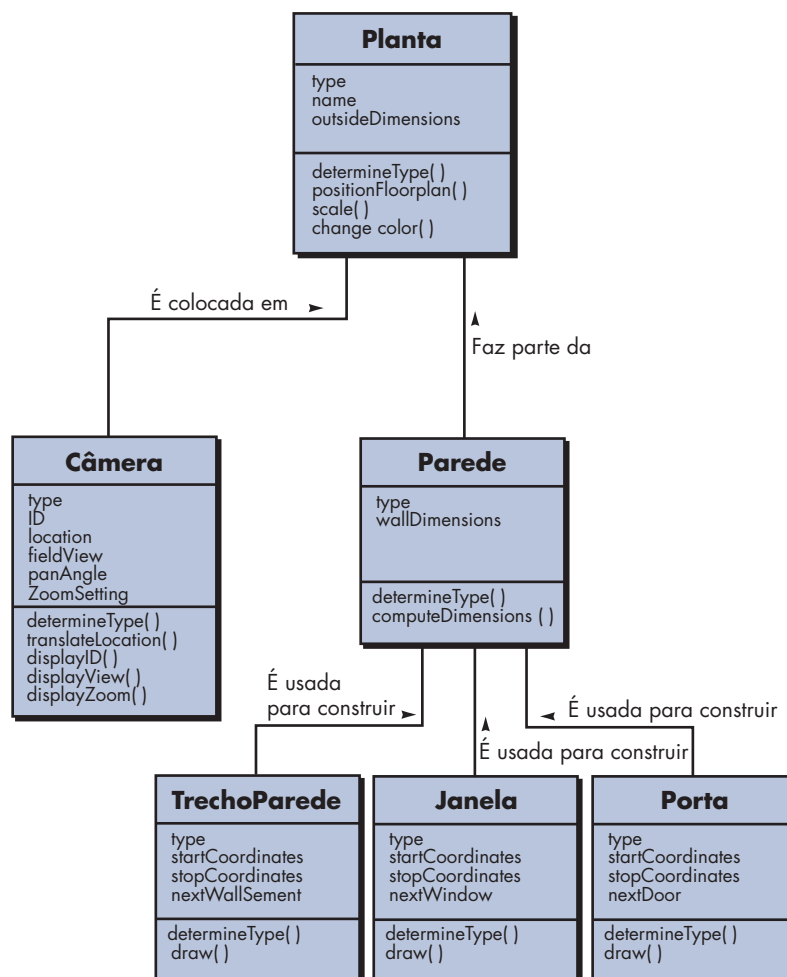
Na realidade, o modelo CRC pode fazer uso de fichas reais ou virtuais. O intuito é desenvolver uma representação organizada das classes. As *responsabilidades* são os atributos e as

“Um dos propósitos dos cartões CRC é fazer com que as falhas apareçam logo, com frequência e de forma barata. É muito mais barato rasgar um monte de cartões do que reorganizar uma grande quantidade de código-fonte.”

C. Horstmann

FIGURA 6.10

Diagrama de classes para Planta (veja o quadro de discussão)



operações que são relevantes para a classe. Em outras palavras, responsabilidade é “qualquer coisa que a classe sabe ou faz” [Amb95]. *Colaboradores* são aquelas classes que são necessárias para fornecer a uma classe as informações necessárias para completar uma responsabilidade. Em geral, *colaboração* implica uma solicitação de informações ou uma solicitação de alguma ação.

Um cartão CRC simples para a classe **Planta** é ilustrado na Figura 6.11. A lista de responsabilidades mostrada no cartão CRC é preliminar e sujeita a acréscimos ou modificações. As classes **Parede** e **Câmera** são indicadas ao lado da responsabilidade que irá requerer sua colaboração.

WebRef

Uma excelente discussão sobre esses tipos de classes pode ser encontrada em www.theumlcafe.com/a0079.htm.

“Os objetos podem ser classificados cientificamente em três grandes categorias: aqueles que não funcionam, aqueles que quebram e aqueles que se perdem.”

Russell Baker

Classes. Foram apresentadas diretrizes básicas para a identificação de classes e objetos no início deste capítulo. A taxonomia dos tipos de classe da Seção 6.5.1 pode ser estendida considerando-se as seguintes categorias:

- *Classes de entidades*, também chamadas *classes de modelo* ou *de negócio*, são extraídas diretamente do enunciado do problema (por exemplo, **Planta** e **Sensor**). Estas representam, tipicamente, coisas que devem ser armazenadas em um banco de dados e persistem ao longo de toda a existência da aplicação (a menos que sejam especificamente eliminadas).
- *Classes de fronteira* são usadas para criar a interface (por exemplo, tela interativa ou relatórios impressos) que o usuário vê e interage à medida que o software é usado. Os objetos de entidades contêm informações importantes para os usuários, mas eles não são exibidos. As *classes de fronteira* são desenvolvidas com a responsabilidade de gerenciar a forma através da qual os objetos de entidades são representados para os usuários. Por exemplo, uma classe de fronteira chamada **JanelaCâmera** teria a responsabilidade de exibir imagens de câmeras de vigilância do sistema *CasaSegura*.
- *Classes de controle* gerenciam uma “unidade de trabalho” [UML03] do início ao fim. Isto é, as classes de controle podem ser desenvolvidas para gerenciar: (1) a criação ou a atualização de objetos de entidades, (2) a instanciação de objetos de fronteira à medida que forem obtendo informações dos objetos de entidades, (3) comunicação complexa entre conjuntos de objetos, (4) validação de dados transmitidos entre objetos ou entre o usuário e a aplicação. Em geral, as classes de controle não são consideradas até que a atividade de projeto tenha sido iniciada.

FIGURA 6.11

Um modelo de cartão CRC

Classe: Planta	
Descrição	
Responsabilidade:	Colaborador:
Incorpora paredes, portas e janelas	
Mostra a posição das câmeras de vídeo	
Define o nome/tipo da planta	
Gerencia o posicionamento na planta	
Amplia/reduz a planta para exibição	
Incorpora paredes, portas e janelas	Parede
Mostra a posição das câmeras de vídeo	Câmera

 Quais diretrizes podem ser aplicadas para alocar responsabilidades às classes?

Responsabilidades. Diretrizes básicas para a identificação das responsabilidades (atributos e operações) foram apresentadas nas Seções 6.5.2 e 6.5.3. Wirfs-Brock e seus colegas [Wir90] sugerem cinco diretrizes para alocação de responsabilidades às classes:

1. A inteligência do sistema deve ser distribuída pelas classes para melhor atender às necessidades do problema. Toda aplicação engloba certo grau de inteligência; aquilo que o sistema sabe e aquilo que ele pode fazer. Essa inteligência pode ser distribuída pelas classes em uma série de maneiras. Classes “burras” (aquelas com poucas responsabilidades) podem ser modeladas para atuar como serventes para algumas poucas classes “inteligentes” (aquelas com muitas responsabilidades). Embora essa abordagem faça com que o fluxo de controle em um sistema se torne simples, ela apresenta algumas desvantagens: concentra toda a inteligência em algumas poucas classes, tornando as mudanças mais difíceis e tende a exigir mais classes e, portanto, um esforço de desenvolvimento maior.

Se a inteligência do sistema for distribuída de forma mais homogênea pelas classes de uma aplicação, cada objeto conhecerá e fará apenas algumas poucas coisas (que em geral são bem focadas), a coesão do sistema aumentará.¹⁷ Isso aumenta a facilidade de manutenção do software e reduz o impacto dos efeitos colaterais devido a mudanças.

Para determinar se a inteligência do sistema está distribuída de maneira apropriada, as responsabilidades indicadas em cada cartão CRC modelo devem ser avaliadas para determinar se alguma classe tem uma lista extraordinariamente longa de responsabilidades. Isso indica uma concentração de inteligência.¹⁸ Além disso, as responsabilidades para cada classe deveriam exibir o mesmo nível de abstração. Por exemplo, entre as operações listadas para uma classe agregada denominada **ContaCorrente** um revisor indica duas responsabilidades: *verificar-saldo-da-conta* e *dar-baixa-cheques-compensados*. A primeira operação (responsabilidade) implica um complexo procedimento lógico e matemático. A segunda é uma atividade administrativa simples. Como essas duas operações não se encontram no mesmo nível de abstração, *dar-baixa-cheques-compensados* deve ser colocada dentro das responsabilidades de **EntradaCheques**, uma classe que é englobada pela classe agregada **ContaCorrente**.

- 2. Cada responsabilidade deve ser declarada da forma mais genérica possível.** Essa diretriz implica que as responsabilidades gerais (tanto atributos quanto operações) devem estar no topo da hierarquia de classes (por elas serem genéricas, serão aplicáveis a todas as subclasses).
- 3. As informações e o comportamento relativos a elas devem residir na mesma classe.** Isso atende o princípio da orientação a objetos denominado *encapsulamento*. Os dados e os processos que manipulam os dados devem ser empacotados como uma unidade coesiva.
- 4. As informações sobre um item devem estar em uma única classe e não distribuída por várias classes.** Uma única classe deve assumir a responsabilidade pelo armazenamento e manipulação de um tipo de informação específico. Tal responsabilidade não deve, em geral, ser compartilhada por uma série de classes. Se as informações forem distribuídas, o software se torna mais difícil de ser mantido e testado.
- 5. Quando apropriado, as responsabilidades devem ser compartilhadas entre classes relacionadas.** Há muitos casos em que uma série de objetos relacionados deve apresentar o mesmo comportamento ao mesmo tempo. Como exemplo, consideremos um videogame que precise exibir as seguintes classes: **Jogador**, **CorpoJogador**, **BraçosJogador**, **PernasJogador**, **CabeçaJogador**. Cada uma dessas classes possui seus próprios atributos (por exemplo,

¹⁷ Coesão é um conceito de projeto que será discutido no Capítulo 8.

¹⁸ Em tais casos, talvez seja necessário subdividir a classe em várias classes ou subsistemas completos para distribuir a inteligência de forma mais eficaz.

posição, orientação, cor, velocidade) e todas têm de ser atualizadas e exibidas à medida que o usuário manipula um *joystick*. Consequentemente, as responsabilidades *atualizar()* e *exibir()* devem ser compartilhadas por cada um dos objetos citados. **Jogador** sabe quando algo mudou e *atualizar()* se faz necessária. Ela colabora com os demais objetos para atingir uma nova posição ou orientação, porém cada objeto controla sua própria visualização.

Colaborações. As classes cumprem suas responsabilidades de duas formas: (1) Uma classe pode usar suas próprias operações para manipular seus próprios atributos, cumprindo, portanto, determinada responsabilidade ou (2) uma classe pode colaborar com outras classes. Wirfs-Brock e seus colegas [Wir90] definem as colaborações da seguinte forma:

As colaborações representam solicitações de um cliente a um servidor no cumprimento de uma responsabilidade do cliente. A colaboração é a expressão do contrato entre o cliente e o servidor... Dizemos que um objeto colabora com outro objeto se, para cumprir uma responsabilidade, ele precisar enviar ao outro objeto quaisquer mensagens. Uma colaboração simples flui em uma direção — representando uma solicitação do cliente ao servidor. Do ponto de vista do cliente, cada uma de suas colaborações está associada a determinada responsabilidade implementada pelo servidor.

As colaborações são identificadas determinando se uma classe pode ou não cumprir cada responsabilidade por si só. Caso não possa, ela precisa interagir com outra classe. Daí, a colaboração.

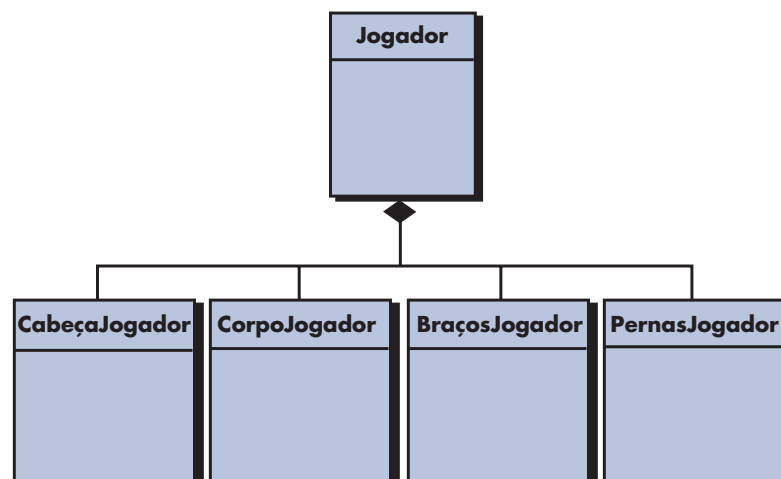
Como exemplo, consideremos a *função de segurança domiciliar do CasaSegura*. Como parte do procedimento de ativação, o objeto **PaineldeControle** deve determinar se algum sensor está aberto. A responsabilidade chamada *determinar-estado-sensor()* é definida. Se existirem sensores abertos, **PaineldeControle** tem de ativar um atributo de estado para “não preparado”. As informações dos sensores podem ser adquiridas de cada objeto **Sensor**. Consequentemente, a responsabilidade *determinar-estado-sensor()* pode ser cumprida apenas se **PaineldeControle** trabalhar em colaboração com **Sensor**.

Para ajudarmos na identificação dos colaboradores, podemos examinar três relacionamentos genéricos diferentes entre as classes [Wir90]: (1) o relacionamento *é-parte-de*, (2) o relacionamento *tem-conhecimento-de* e (3) o relacionamento *depende-de*. Cada um desses três relacionamentos genéricos é considerado remidamente nos parágrafos a seguir.

Todas as classes que fazem parte de uma classe agregada são interligadas à classe agregada por meio de um relacionamento *é-parte-de*. Considerando as classes definidas para o videogame citado anteriormente, a classe **CorpoJogador** *faz-parte-do* **Jogador**, **assim como** **BraçosJogador**, **PernasJogador** e **CabeçaJogador**. Em UML, tais relacionamentos são representados na forma da agregação mostrada na Figura 6.12.

FIGURA 6.12

Uma classe composta de agregação



Quando uma classe tem de adquirir informações de outra classe, é estabelecido o relacionamento *tem-conhecimento-de*. A responsabilidade *determinar-estado-sensor()* citada anteriormente é um exemplo de um relacionamento *tem-conhecimento-de*.

A relação *depende-de* implica que duas classes têm uma dependência que não é conseguida por *tem-conhecimento-de* ou *é-parte-de*. Por exemplo, **CabeçaJogador** sempre tem de estar interligado ao **CorpoJogador** (a menos que o videogame seja particularmente violento), embora cada objeto pudesse existir sem o conhecimento direto do outro. Um atributo do objeto **CabeçaJogador** chamado **posição-central** é determinado a partir da posição central do **CorpoJogador**. Essa informação é obtida através de um terceiro objeto, **Jogador**, que a adquire de **CorpoJogador**. Portanto, **CabeçaJogador** *depende-do* **CorpoJogador**.

Em todos os casos, o nome da classe colaboradora é registrado no cartão CRC modelo, próximo à responsabilidade que gerou a colaboração. Consequentemente, o cartão contém uma lista de responsabilidades e as colaborações correspondentes que permitem que as responsabilidades sejam cumpridas (Figura 6.11).

Quando um modelo CRC completo tiver sido desenvolvido, os interessados poderão revisar o modelo usando a seguinte abordagem [Amb95]:

1. Todos os participantes da revisão (do modelo CRC) recebem um subconjunto dos cartões CRC. Os cartões que colaboram devem ser separados (nenhum revisor deve ter dois cartões que colaboram).
2. Todos os cenários de uso (e diagramas de casos de uso correspondentes) devem ser organizados em categorias.
3. O líder da revisão lê o caso de uso pausadamente. À medida que o líder da revisão chega a um objeto com nome, ele passa uma ficha para a pessoa que está com o cartão da classe correspondente. Por exemplo, um caso de uso para o *CasaSegura* conteria a seguinte narrativa:

O proprietário do imóvel observa o painel de controle do *CasaSegura* para determinar se o sistema está pronto para operar. Se o sistema não estiver pronto, o proprietário do imóvel deve fechar manualmente as janelas/portas de modo que o indicador pronto esteja presente. [Um indicador não preparado implica que um sensor está aberto, isto é, que uma porta ou janela está aberta.]

Quando o líder da revisão chega em “painel de controle”, na narrativa de caso de uso, a ficha é passada para a pessoa que está com o cartão **PaineldeControle**. A frase “implica que um sensor está aberto” requer que o cartão contenha a responsabilidade que irá validar esta implicação (a responsabilidade *determinar-estado-sensor()* realiza isso). Próximo à responsabilidade no cartão se encontra o colaborador **Sensor**. A ficha é então passada para o objeto **Sensor**.

4. Quando a ficha é passada, solicita-se ao portador do cartão **Sensor** que descreva as responsabilidades anotadas no cartão. O grupo determina se uma (ou mais) das responsabilidades satisfaz à necessidade do caso de uso.
5. Se as responsabilidades e colaborações anotadas nos cartões não puderem satisfazer ao caso de uso, as modificações são feitas nos cartões. Estas podem incluir a definição das novas classes (e os cartões CRC correspondentes) ou a especificação das responsabilidades novas ou revisadas ou das colaborações em cartões existentes.

Esse *modus operandi* prossegue até que o caso de uso seja finalizado. Quando todos os casos de uso tiverem sido revistos, a modelagem de requisitos continua.

CASA SEGURA



Modelos CRC

Cena: Sala do Ed, quando se inicia a modelagem de requisitos.

Atores: Jamie, Vinod e Ed — todos os membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

[Vinod decidiu mostrar a Ed como desenvolver cartões CRC dando-lhe um exemplo.]

Vinod: Enquanto você trabalhava na função de vigilância e Jamie estava envolvido com a função de segurança, trabalhei na função de administração domiciliar.

Ed: E em que ponto você se encontra? O Marketing vive mudando de ideia.

Vinod: Eis o caso de uso preliminar para toda a função... Nós o refinamos um pouco, mas isso deve lhe dar uma visão geral...

Caso de uso: A função de administração domiciliar do *CasaSegura*.

Narrativa: Queremos usar a interface de administração domiciliar em um PC ou via Internet para controlar dispositivos eletrônicos que possuem controladores de interface sem fio. O sistema deve permitir que eu ligue e desligue luzes específicas, controle aparelhos que estiverem conectados a uma interface sem fio, configure o meu sistema de aquecimento e ar-condicionado para as temperaturas que eu desejar. Para tanto, quero escolher os dispositivos com base na planta da casa. Cada dispositivo tem de ser identificado na planta da casa. Como recurso opcional, quero controlar todos os dispositivos audiovisuais — áudio, televisão, DVD, gravadores digitais e assim por diante. Com uma única seleção, quero ser capaz de configurar a casa inteira para várias situações. Uma delas é *em casa*, a outra é *fora de casa*, a terceira é *viagem de fim de semana* e a quarta é *viagem prolongada*. Todas essas situações terão ajustes de configuração aplicados a todos os dispositivos. Nos estados *viagem de fim de semana* e *viagem prolongada*, o sistema deve acender e apagar as luzes da casa em intervalos aleatórios (para parecer que alguém está em casa) e controlar o sistema de aquecimento e ar-condicionado. Devo ser capaz de cancelar essas configurações via Internet com a proteção de uma senha apropriada...

Ed: O pessoal do hardware já bolou todas as interfaces sem fio?

Vinod (sorrindo): Eles estão trabalhando nisso; digamos que não há nenhum problema. De qualquer forma, extraí um monte de classes para a administração domiciliar e podemos usar uma delas como exemplo.

Usemos a classe **InterfaceAdministraçãoDomiciliar**.

Ed: OK... Portanto, as responsabilidades são... Os atributos e as operações para a classe, e as colaborações são as classes para as quais as responsabilidades apontam.

Vinod: Pensei que você não entendesse sobre CRC.

Ed: Um pouquinho, mas vá em frente.

Vinod: Portanto, eis minha definição de classe para **InterfaceAdministraçãoDomiciliar**.

Atributos:

optionsPanel — contém informações sobre os botões que permitem ao usuário escolher a funcionalidade.

situationPanel — contém informações sobre os botões que permitem ao usuário escolher a situação.

floorplan — o mesmo que o objeto de vigilância, porém este aqui mostra os dispositivos.

deviceIcons — informações sobre os ícones representando luzes, aparelhos, HVAC etc.

devicePanels — simulação do painel de controle de dispositivos ou de aparelhos; possibilita o controle.

Operações:

displayControl(), selectControl(), displaySituation(), select situation(), accessFloorplan(), selectDeviceIcon(), displayDevicePanel(), accessDevicePanel()...

Classe: InterfaceAdministraçãoDomiciliar

Responsabilidade

displayControl()
selectControl()
displaySituation()
selectSituation()
accessFloorplan()
...

Colaborador

PaineldeOpção (classe)
PaineldeOpção (classe)
PaineldeSituação (classe)
PaineldeSituação (classe)
Planta (classe)...

Ed: Portanto, quando a operação *accessFloorplan()* for chamada, ela colaborará com o objeto **FloorPlan** exatamente com aquela que desenvolvemos para a função de vigilância. Espere, tenho uma descrição dela aqui comigo. (Eles observam a Figura 6.10.)

Vinod: Exatamente. E se quiséssemos rever o modelo de classes inteiro, poderíamos começar com este cartão, depois ir para o cartão do colaborador e a partir deste ponto para um dos colaboradores do colaborador, e assim por diante.

Ed: Uma boa forma de descobrir omissões ou erros.

Vinod: Isso aí.

6.5.5 Associações e dependências

Em muitos casos, duas classes de análise são relacionadas entre si de alguma maneira, de modo muito parecido como dois objetos de dados poderiam estar relacionados entre si (Seção 6.4.3). Na UML essas relações são chamadas *associações*. Referindo-se novamente à Figura 6.10, a classe **Planta** é definida identificando-se um conjunto de associações entre **Planta** e duas outras classes, **Câmera** e **Parede**. A classe **Parede** é associada a três classes que permitem que uma parede seja construída, **TrechoParede**, **Janela** e **Porta**.

PONTO-CHAVE

A associação define um relacionamento entre classes. A multiplicidade define quantos de uma classe estão relacionados com quantos de uma outra classe.

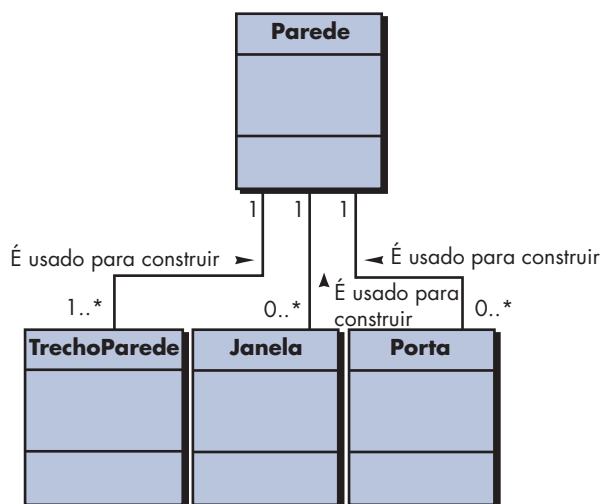
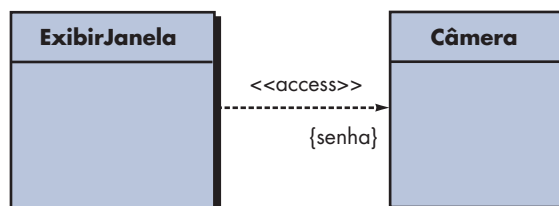


O que é um estereótipo?

Em alguns casos, uma associação pode ser mais bem definida indicando-se a *multiplicidade*. Referindo-se à Figura 6.10, um objeto **Parede** é construído por meio de um ou mais objetos **TrechoParede**. Além disso, o objeto **Parede** pode conter nenhum ou alguns objetos **Janela** e nenhum ou alguns objetos **Porta**. Essas restrições de multiplicidade são ilustradas na Figura 6.13, em que “um ou mais” é representado usando-se 1..* e “zero ou mais” por 0..*. Na UML, o asterisco indica um limite superior infinito no intervalo.¹⁹

Em muitos casos, existe um relacionamento cliente/servidor entre duas classes de análise. Em tais casos, uma classe cliente depende da classe servidora de alguma forma e se estabelece um relacionamento *relação de dependência*. Dependências são definidas por um estereótipo. *Estereótipo* é um “mecanismo de extensibilidade” [Ar102] dentro da UML que nos permite definir um elemento de modelagem especial cuja semântica é definida de forma personalizada. Na UML os estereótipos são representados entre << >> (por exemplo, <<stereotype>>).

Como exemplo de uma dependência simples no sistema de vigilância *CasaSegura*, um objeto **Câmera** (neste caso, a classe servidora) fornece uma imagem de vídeo para um objeto **ExibirJanela** (neste caso, a classe cliente). O relacionamento entre esses dois objetos não é uma associação simples, embora não exista uma associação de dependência. Em um caso de uso escrito para vigilância (não mostrado), você toma conhecimento de que uma senha especial tem de ser fornecida para visualizar posições de câmera específicas. Uma maneira de conseguir isso é fazer com que **Câmera** solicite uma senha e então dê permissão para **ExibirJanela** produzir a exibição de vídeo. Isso pode ser representado como mostra a Figura 6.14 em que <<access>> implica que o uso da saída de câmera é controlado por uma senha especial.

FIGURA 6.13**Multiplicidade****FIGURA 6.14****Dependências**

¹⁹ Outros relacionamentos de multiplicidade — um para um, um para vários, vários para vários, um para um intervalo especificado com limites inferior e superior e outros — podem ser indicadas como parte de uma associação.

PONTO-CHAVE

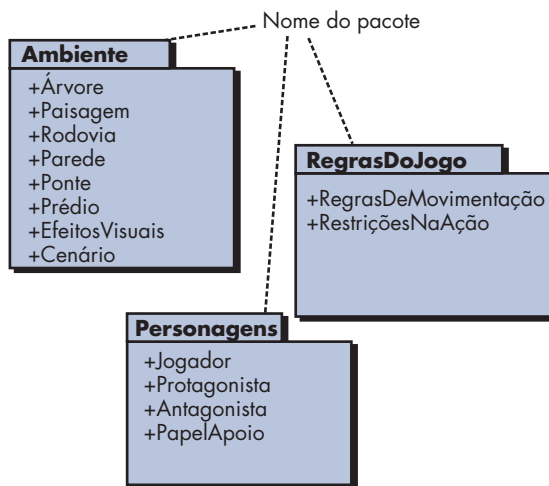
Um pacote é usado para montar um conjunto de classes relacionadas.

6.5.6 Pacotes de análise

Uma importante parte da modelagem de análise é a categorização. Vários elementos do modelo de análise (por exemplo, casos de uso, classes de análise) são categorizados de modo que são empacotados como um agrupamento — denominado *pacote de análise* — que recebe um nome representativo.

Para ilustrarmos o uso de pacotes de análise, consideremos o videogame introduzido anteriormente. À medida que o modelo de análise para o videogame é desenvolvido, um número maior de classes é extraído. Algumas focalizam o ambiente do jogo — as cenas que o usuário vê à medida que o jogo vai se desenrolando. Classes como **Árvore**, **Paisagem**, **Rodovia**, **Parede**, **Ponte**, **Prédio** e **EfeitoVisual** poderiam cair dentro desta categoria. Outras focalizam os personagens do jogo, descrevendo suas características físicas, ações e restrições. Classes como **Jogador** (descrito anteriormente), **Protagonista**, **Antagonista** e **PapéisApoio** poderiam ser definidas. Outras ainda descrevem as regras do jogo — como um jogador navega pelo ambiente. Classes como **RegrasDeMovimentação** e **RestriçõesNaAção** são candidatas aqui. Poderiam existir muitas outras categorias. Essas classes podem ser agrupadas em pacotes de análise conforme mostra a Figura 6.15.

O sinal de mais antes do nome da classe de análise em cada pacote indica que as classes têm visibilidade pública e são, conseqüentemente, acessíveis de outros pacotes. Embora não sejam mostrados na figura, outros símbolos podem anteceder um elemento dentro de um pacote. Um sinal de menos indica que um elemento está oculto de todos os demais pacotes e um símbolo # indica que um elemento é acessível apenas para pacotes contidos em determinado pacote.

FIGURA 6.15**Pacotes****6.6 RESUMO**

O objetivo da modelagem de requisitos é criar uma variedade de representações que descrevem aquilo que o cliente requer, estabelece uma base para a criação de um projeto de software e define um conjunto de requisitos que podem ser validados assim que o software for construído. O modelo de requisitos preenche a lacuna entre uma representação sistêmica que descreve o sistema como um todo ou a funcionalidade de negócio e um projeto de software que descreve a arquitetura da aplicação de software, a interface do usuário e a estrutura no nível de componentes.

Os modelos baseados em cenário representam requisitos de software sob o ponto de vista do usuário. O caso de uso — uma narrativa ou descrição dirigida por modelos de uma interação entre um ator e o software — é o principal elemento da modelagem. Obtido durante o levantamento de requisitos, o caso de uso define as etapas fundamentais para uma função ou interação específica. O grau de formalidade dos casos de uso e detalhes varia, porém o resultado final fornece a entrada necessária para as demais atividades de modelagem de análise. Os cenários também podem ser descritos usando-se um diagrama de atividades — uma representação gráfica do tipo fluxograma que represente o fluxo de processamento dentro de um cenário específico. Os diagramas de raia ilustram como o fluxo de processamento é alocado a vários atores ou classes.

A modelagem de dados é usada para descrever o espaço de informações que serão construídas ou manipuladas pelo software. A modelagem de dados começa pela representação dos objetos de dados — informações compostas que devem ser compreendidas pelo software. Os atributos de cada objeto de dados são identificados e os relacionamentos entre objetos de dados, descritos.

A modelagem baseada em classes usa informações extraídas dos elementos de modelagem de dados baseados em cenários para identificar as classes de análise. Uma análise sintática pode ser usada para extrair classes, atributos e operações candidatos com base em narrativas textuais. São definidos os critérios para a definição de uma classe. Um conjunto de cartões classe-responsabilidade-colaborador pode ser usado para definir relações entre classes. Além disso, uma variedade de notação da modelagem UML pode ser aplicada para definir hierarquias, relacionamentos, associações, agregações e dependências entre as classes. Pacotes de análise são usados para categorizar e agrupar classes de maneira que as tornem mais administráveis para grandes sistemas.

PROBLEMAS E PONTOS A PONDERAR

- 6.1. Existe a possibilidade de começar a codificar logo depois de um modelo de análise ter sido criado? Justifique sua resposta e, em seguida, argumente ao contrário.
- 6.2. Uma regra prática para análise é que o modelo “deve se concentrar nos requisitos visíveis dentro do domínio do negócio ou problema”. Que tipos de requisitos *não* são visíveis nesses domínios? Forneça alguns exemplos.
- 6.3. Qual o propósito da análise de domínio? Como está relacionada com o conceito de padrões de requisitos?
- 6.4. É possível desenvolver um modelo de análise efetivo sem desenvolver todos os quatro elementos da Figura 6.3? Explique.
- 6.5. Foi-lhe solicitado construir um dos seguintes sistemas:
 - a. um sistema de matrícula em cursos baseado em rede para a sua universidade.
 - b. um sistema de processamento de pedidos baseado na Web para uma loja de informática.
 - c. um sistema de faturas simples para um pequeno negócio.
 - d. um livro de receitas baseado na Internet que é embutido em forno elétrico ou micro-ondas.

Escolha o sistema de seu interesse e desenvolva um diagrama entidade-relação que descreva objetos de dados, relacionamentos e atributos.

- 6.6. O departamento de obras públicas de uma grande cidade decidiu desenvolver um sistema de tapa-buracos (*pothole tracking and repair system*, PHTRS) baseado na Web. Segue uma descrição:

Os cidadãos podem entrar em um site e relatar o local e a gravidade dos buracos. À medida que são relatados, os buracos são registrados em um “sistema de reparos do departamento de obras públicas” e recebem um número identificador, armazenado pelo endereço (nome da rua), tamanho (em

uma escala de 1 a 10), localização (no meio da rua, meio-fio etc.), bairro (determinado com base no endereço) e prioridade para o reparo (determinada segundo o tamanho do buraco). Os dados de solicitação de trabalho são associados a cada buraco e incluem a localização e o tamanho do buraco, equipe de obras identificando o número, o número de operários da equipe, equipamento alocado, horas usadas para o reparo, estado do buraco (trabalho em andamento, reparado, reparo temporário, não reparado), quantidade de material de preenchimento utilizado e custo do reparo (calculado com base nas horas utilizadas, no número de pessoas, no material e no equipamento usado). Por fim, é criado um arquivo de danos para armazenar informações sobre o dano relatado devido ao buraco e que inclui o nome, endereço e telefone do cidadão, tipo de dano e custo monetário do dano. O PHTRS é um sistema on-line; todas as consultas devem ser feitas interativamente.

- a. Desenhe um diagrama de caso de uso UML para o sistema PHTRS. Você terá de fazer uma série de suposições sobre a maneira através da qual um usuário interage com esse sistema.
 - b. Desenvolva um modelo de classes para o sistema PHTRS.
- 6.7.** Escreva um caso de uso baseado em modelo para o sistema de administração domiciliar *CasaSegura* descrito informalmente no quadro após a Seção 6.5.4.
- 6.8.** Desenvolva um conjunto completo de cartões CRC sobre o produto ou sistema que você escolher como parte do Problema 6.5.
- 6.9.** Realize uma revisão dos cartões CRC com seus colegas. Quantas classes, responsabilidades e colaboradores adicionais são acrescentados como consequência da revisão?
- 6.10.** O que é um pacote de análise e como poderia ser usado?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Os casos de uso podem servir como base para todas as abordagens de modelagem de requisitos. O tema é discutido de forma abrangente por Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander e Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Bittner e Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [Coc01b] e outras referências citadas nos Capítulos 5 e 6.

A modelagem de dados apresenta um útil método para examinar o espaço de informações. Livros como os de Hoberman [Hob06] e Simsion e Witt [Sim05] são tratados relativamente completos. Além disso, Allen e Terry (*Beginning Relational Data Modeling*, 2. ed., Apress, 2005), Allen (*Data Modeling for Everyone*, Wrox Press, 2002), Teorey e seus colegas (*Database Modeling and Design: Logical Design*, 4. ed., Morgan Kaufmann, 2005) e Carlis e Maguire (*Mastering Data Modeling*, Addison-Wesley, 2000) trazem tutoriais detalhados para criação de modelos de dados de alta qualidade. Um interessante livro de Hay (*Data Modeling Patterns*, Dorset House, 1995) lista padrões de modelos de dados típicos encontrados em diversas áreas de aplicação.

As técnicas de modelagem UML que podem ser aplicadas tanto na análise quanto no projeto são discutidos por O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*, Wiley, 2005), Arlow e Neustadt (*UML 2 and the Unified Process*, 2. ed., Addison-Wesley, 2005), Roques (*UML in Practice*, Wiley, 2004), Dennis e seus colegas (*Systems Analysis and Design with UML Version 2.0*, Wiley, 2004), Larman (*Applying UML and Patterns*, 2. ed., Prentice-Hall, 2001) e Rosenberg e Scott (*Use Case Driven Object Modeling with UML*, Addison-Wesley, 1999).

Uma ampla gama de fontes de informação sobre modelagem de requisitos se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevantes à modelagem de requisitos, pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

MODELAGEM DE REQUISITOS: FLUXO, COMPORTAMENTO, PADRÕES E APLICAÇÕES BASEADAS NA WEB (WEBAPP)

CONCEITOS-CHAVE

diagramas de sequência	191
especificação de processo	186
modelo comportamental ..	188
modelo de configuração ..	200
modelo de conteúdo ...	198
modelo de controle de fluxo	184
modelo de fluxo de dados	182
modelo de interação ..	200
modelo de navegação ..	208
modelo funcional	200
padrões de análise ...	193
WebApps	197

Após nossa discussão de casos de uso, modelagem de dados e modelos baseados em classes no Capítulo 6, é razoável perguntar: “Não seriam essas representações de modelagem de requisitos suficientes?”.

A única resposta razoável é: “Depende”. Para alguns tipos de software, o caso de uso poderia ser a única representação de modelagem de requisitos exigida. Para outros, é escolhida uma abordagem orientada a objetos, e modelos baseados em classes poderiam ser desenvolvidos. Porém, em outras situações, requisitos de aplicação complexos poderiam demandar um exame de como os objetos de dados são transformados à medida que fluem por um sistema; como uma aplicação se comporta como consequência de eventos externos; se o conhecimento do domínio existente pode ser adaptado ao problema atual; ou no caso de sistemas e aplicações baseadas na Web (WebApp), como o conteúdo e a funcionalidade combinados poderiam dar a um usuário final a habilidade de navegar com êxito por uma WebApp para atingir essas metas.

PANORAMA

O que é? O modelo de requisitos possui várias dimensões diferentes. No presente capítulo você aprenderá modelos orientados a fluxos, modelos comportamentais e considerações sobre análise de requisitos especiais que entram em cena quando são desenvolvidas aplicações baseadas na Web (WebApps). Cada uma dessas representações de modelagem complementa os casos de uso, os modelos de dados e os modelos baseados em classes discutidos no Capítulo 6.

Quem realiza? Um engenheiro de software (às vezes denominado “analista”) constrói o modelo usando os requisitos extraídos de vários interessados.

Por que é importante? Sua visão sobre os requisitos do software aumenta em proporção direta ao número de diferentes dimensões da modelagem de requisitos. Embora talvez você não tenha tempo, recursos ou inclinação para desenvolver cada representação sugerida neste capítulo e no Capítulo 6, deve reconhecer que cada abordagem de modelagem lhe dá uma forma diferente de visualizar o problema. Como consequência, você (e outros interessados) estarão mais bem preparados para avaliar se o que deve ser

cumprido foi ou não especificado de maneira apropriada.

Quais são as etapas envolvidas? A modelagem orientada a fluxos dá uma indicação de como os objetos de dados são transformados pelas funções de processamento. A modelagem comportamental representa os estados do sistema e suas classes e o impacto dos eventos sobre esses estados. A modelagem baseada em padrões faz uso do conhecimento do domínio existente para facilitar a análise de requisitos. Os modelos de requisitos para WebApp são especialmente adaptados para a representação das necessidades relacionadas ao conteúdo, interação, função e configuração.

Qual é o artefato? Uma ampla gama de formas textuais e esquemáticas pode ser escolhida para o modelo de requisitos. Cada uma dessas representações dá uma visão de um ou mais dos elementos do modelo.

Como garantir que o trabalho foi realizado corretamente? Os artefatos do modelamento de requisitos devem ser revisados em termos de correção, completude e consistência. Devem refletir os requisitos de todos os interessados e estabelecer uma base a partir da qual o projeto pode ser conduzido.

7.1 ESTRATÉGIAS DE MODELAGEM DE REQUISITOS

Uma visão da modelagem de requisitos, denominada *análise estruturada*, considera os dados e os processos que os transformam entidades distintas. Os objetos de dados são modelados de maneira que defina seus atributos e relacionamentos. Processos que manipulam objetos de dados são modelados para mostrar como transformam os dados à medida que objetos de dados fluem através do sistema. A segunda abordagem para a modelagem de análise denominada *análise orientada a objetos*, enfoca a definição de classes e a maneira pela qual colaboram entre si para atender os requisitos do cliente.

Embora o modelo de análise que propomos neste livro combina recursos de ambas as abordagens, as equipes de software em geral optam por uma abordagem e excluem todas as representações da outra. A questão não é qual a melhor, mas sim, qual combinação de representações irá fornecer aos interessados o melhor modelo de requisitos de software e a ligação mais efetiva para o projeto de software.

7.2 MODELAGEM ORIENTADA A FLUXOS



Alguns poderão sugerir que o DFD seja coisa do passado e que não tem mais lugar na prática moderna. Trata-se de uma visão que exclui um modo de representação potencialmente útil no nível de análise. Se ele for útil para comunicar e documentar, use o DFD.

Embora a modelagem orientada a fluxos de dados seja vista como uma técnica ultrapassada por alguns engenheiros de software, continua a ser uma das notações de análise de requisitos mais largamente usadas hoje em dia.¹ Embora o *diagrama de fluxo de dados* (*data flow diagram*, DFD) e diagramas relacionados não sejam uma parte formal da UML, podem ser usados para complementar os diagramas UML e darem uma visão adicional sobre o fluxo e os requisitos do sistema.

O DFD adota uma visão entrada-processo-saída de um sistema. Isto é, os objetos de dados entram no software, são transformados por elementos de processamento e os objetos de dados resultantes saem do software. Os objetos de dados são representados por setas rotuladas e as transformações, por círculos (também chamados bolhas). O DFD é apresentado de uma forma hierárquica: o primeiro modelo de fluxo de dados (algumas vezes denominado DFD nível 0 ou *diagrama de contexto*) representa o sistema como um todo. Diagramas de fluxo de dados subsequentes refinam o diagrama de contexto, fornecendo detalhamento progressivo em cada nível subsequente.

7.2.1 Criação de um modelo de fluxo de dados

O diagrama de fluxo de dados permite que desenvolvamos modelos do domínio de informações e domínio funcional. À medida que o DFD é refinado com níveis de detalhe cada vez maiores, realizamos uma decomposição funcional implícita do sistema. Ao mesmo tempo, o refinamento do DFD resulta em um correspondente refinamento dos dados à medida que se avança nos processos que constituem a aplicação.

Algumas diretrizes simples podem ajudar muito durante a obtenção do diagrama de fluxo de dados: (1) o diagrama de fluxo de dados nível 0 deve representar o software/sistema como uma única bolha; (2) as entradas e saídas primárias devem ser cuidadosamente indicadas; (3) o refinamento deve começar isolando prováveis processos, objetos de dados e repositórios de dados para ser representados no nível seguinte; (4) todas as setas e bolhas devem ser rotuladas com nomes significativos; (5) deve-se manter a *continuidade do fluxo de informações* de nível para nível,² e (6) deve-se refinar uma bolha por vez. Há uma tendência natural de se complicar demais o diagrama de fluxo de dados. Isso acontece quando tentamos mostrar muitos detalhes precocemente ou representar aspectos procedurais do software em vez de fluxo de informações.

"O propósito dos diagramas de fluxo de dados é fornecer uma ponte semântica entre usuários e desenvolvedores de sistemas."

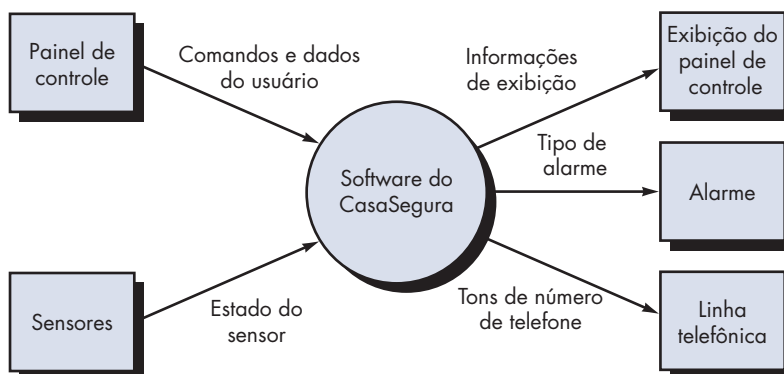
Kenneth Kozar

¹ A modelagem de fluxo de dados é uma atividade de modelagem fundamental na *análise estruturada*.

² Ou seja, os objetos de dados que fluem para dentro do sistema ou qualquer transformação em um nível devem ser os mesmos objetos de dados (ou suas partes constituintes) que fluem para dentro da transformação em um nível mais refinado.

FIGURA 7.1

DFD em nível de contexto para a função de segurança do CasaSegura



PONTO-CHAVE

A continuidade do fluxo de informações deve ser mantida à medida que cada nível DFD seja refinado. Isso significa que entrada e saída em um nível devem ser o mesmo que entrada e saída em um nível refinado.

Para ilustrarmos o uso do DFD e da notação relacionada, consideremos mais uma vez a função de segurança do *CasaSegura*. Um DFD nível 0 para a função de segurança é mostrado na Figura 7.1. As *entidades externas* (retângulos) primárias produzem informações para uso do sistema e consomem informações geradas pelo sistema. As setas rotuladas representam objetos de dados ou hierarquias de objetos de dados. Por exemplo, **dados e comandos de usuário** englobam todos os comandos de configuração, todos os comandos de ativação/desativação, todas as variações de interações e todos os dados introduzidos para qualificar ou expandir um comando.

O DFD nível 0 agora tem de ser expandido para um modelo de fluxo de dados nível 1. Mas como prosseguimos? Seguindo uma abordagem sugerida no Capítulo 6, deveríamos aplicar uma “análise sintática” [Abb83] à narrativa de caso de uso que descreve a bolha no nível de contexto. Isolamos todos os substantivos (e locuções substantivas) e verbos (e locuções verbais) em uma narrativa de processamento do *CasaSegura* obtida durante a primeira reunião para levantamento de necessidades. Recapitulando o texto narrativo de processamento com análise sintática apresentado na Seção 6.5.1:



A análise sintática não é infalível, mas pode proporcionar um excelente passo inicial, caso esteja havendo dificuldades para definir objetos de dados e as transformações que neles operam.

A função de segurança domiciliar do *CasaSegura* permite que o proprietário do imóvel configure o sistema de segurança quando ele é instalado, monitorea todos os sensores conectados ao sistema de segurança e interage com o proprietário do imóvel através da Internet, de um PC ou de um painel de controle.

Durante a instalação, o PC do *CasaSegura* é utilizado para programar e configurar o sistema. São atribuídos um número e um tipo a cada sensor, é programada uma senha-mestra para armar e desarmar o sistema e são introduzidos número(s) de telefone para disparar quando ocorre um evento de sensor.

Quando um evento de sensor é reconhecido, o software aciona um alarme sonoro agregado ao sistema. Após um tempo de espera que é especificado pelo proprietário do imóvel durante as atividades de configuração do sistema, o software disca um número de telefone de um serviço de monitoramento, fornece informações sobre o local, relatando a natureza do evento detectado. O número de telefone será rediscado a cada 20 segundos até que seja completada a ligação telefônica.

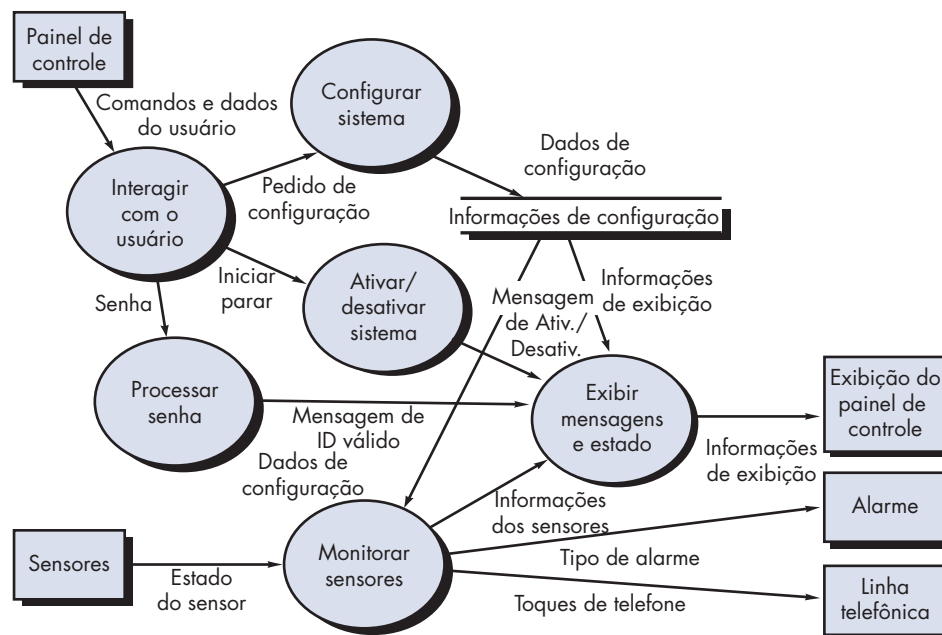
O proprietário do imóvel recebe informações de segurança através de um painel de controle, do PC ou de um navegador, coletivamente denominados de interface. A interface mostra mensagens de aviso bem como informações sobre o estado do sistema no painel de controle, no PC ou na janela do navegador. A interação com o proprietário do imóvel assume a seguinte forma...



Assegure-se de que a narrativa de processamento que você pretende analisar sintaticamente esteja escrita no mesmo nível de abstração por todo o processo.

Referindo-se à análise sintática, os verbos são processos do *CasaSegura* e podem ser representados como bolhas em um DFD subsequente. Os substantivos podem ser entidades externas (retângulos), objetos de controle ou de dados (setas) ou então repositórios de dados (linhas duplas). Da discussão do Capítulo 6, lembre-se de que os substantivos e verbos podem ser associados entre si (por exemplo, são atribuídos um número e um tipo a cada sensor; dessa forma, **número** e **tipo** são atributos do objeto de dados **sensor**). Consequentemente, realizando uma análise sintática na narrativa de processamento para uma bolha em qualquer nível DFD, podemos gerar muitas informações úteis sobre como prosseguir com o refinamento para o próximo nível. Usando essas informações, é mostrado um DFD nível 1 na Figura 7.2. O processo no nível de contexto mostrado na Figura 7.1 foi expandido em seis processos obtidos do exame da análise sintática. De modo

FIGURA 7.2
DFD nível 1 para
a função de
segurança do
CasaSegura

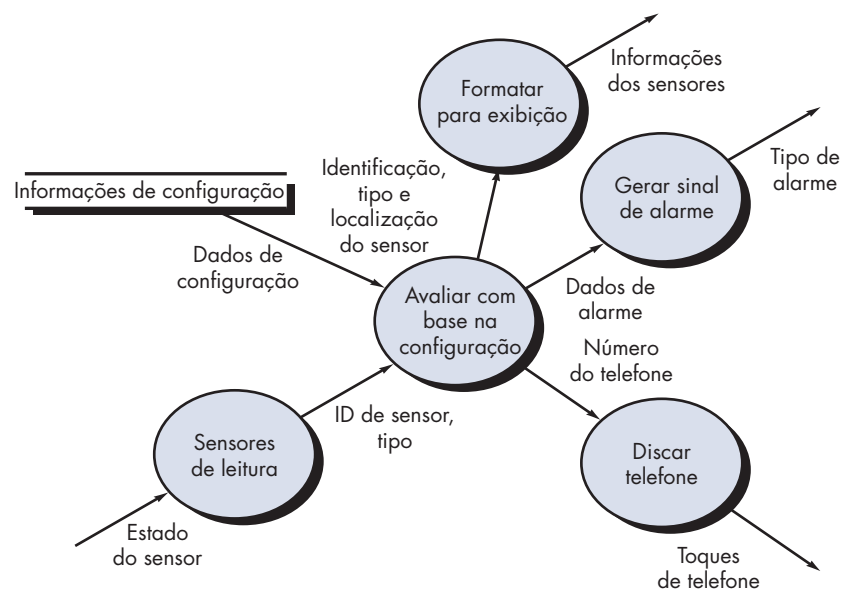


similar, o fluxo de informações entre processos no nível 1 foi extraído da análise sintática. Além disso, a continuidade do fluxo de informações é mantida entre os níveis 0 e 1.

Os processos representados no nível 1 do DFD podem ser mais refinados em níveis mais baixos. Por exemplo, o processo *monitorar sensores* pode ser refinado em um DFD nível 2, conforme mostra a Figura 7.3. Note mais uma vez que a continuidade do fluxo de informações foi mantida entre os níveis.

O refinamento de DFDs prossegue até que cada bolha realize uma simples função. Até que o processo representado pela bolha realize uma função que seria facilmente implementada como componente de um programa. No Capítulo 8, discutiremos um conceito, chamado *coesão*, que pode ser usado para avaliar o foco do processamento de determinada função. Por enquanto, nos esforçaremos para refinar os DFDs até que cada bolha tenha um “único objetivo”.

FIGURA 7.3
DFD nível 2 que
refina o processo
monitorar sensores



7.2.2 Criação de um modelo de fluxo de controle

Para alguns tipos de aplicações, o modelo de dados e o diagrama de fluxo de dados bastam para ter uma visão dos requisitos de software. Entretanto, conforme já citado, um grande grupo de aplicações é “dirigido” por eventos e não por dados, produzem informações de controle em vez de relatórios ou telas e processam informações com grande preocupação com o tempo e desempenho. Tais aplicações requerem o uso da *modelagem de fluxo de controle*, além da modelagem de fluxo de dados.

Já vimos que um item de evento ou controle é implementado como um valor booleano (por exemplo, verdadeiro ou falso, ligado ou desligado, 1 ou 0) ou como uma lista discreta de condições (por exemplo, vazio, bloqueado, cheio). Para escolher eventos possíveis candidatos a eventos, sugerem-se as seguintes diretrizes:

 Como escolher eventos potenciais para um diagrama de fluxo de controle, um diagrama de estados ou uma CSPEC?

- Listar todos os sensores “lidos” pelo software.
- Listar todas as condições de interrupção.
- Listar todas as “chaves” acionadas por um operador.
- Listar todas as condições de dados.
- Recapitular a análise sintática de substantivos/verbos aplicada à narrativa de processamento, revisar todos os “itens de controle” como possíveis entradas/saídas de especificação de controle.
- Descrever o comportamento de um sistema por meio da identificação de seus estados, de como cada estado é atingido e definir as transições entre os estados.
- Concentrar-se em possíveis omissões — um erro muito comum na especificação de controles; por exemplo, perguntar: “Existe alguma outra maneira de chegarmos ou sairmos desse estado?”.

Entre os muitos itens de eventos e de controle que fazem parte do software *CasaSegura* temos **evento de sensor** (isto é, um sensor foi acionado), **flag piscante** (um sinal para fazer com que a exibição fique piscando) e **iniciar/parar chave** (um sinal para ligar ou desligar o sistema).

7.2.3 A especificação de controles

Uma *especificação de controle* (*control specification*, *CSPEC*) representa o comportamento do sistema (no nível em que foi referido) de duas maneiras diferentes.³ A CSPEC contém um diagrama de estados que é uma especificação de comportamento sequencial. Ela também pode conter uma tabela de ativação de programas — uma especificação de comportamento combinatória.

A Figura 7.4 representa um diagrama de estados⁴ preliminar para o modelo de fluxo de controle nível 1 para o *CasaSegura*. O diagrama indica como o sistema responde a eventos à medida que passa pelos quatro estados definidos nesse nível. Através da revisão de diagramas de estados, podemos determinar o comportamento do sistema e, mais importante ainda, determinar se há “furos” no comportamento especificado.

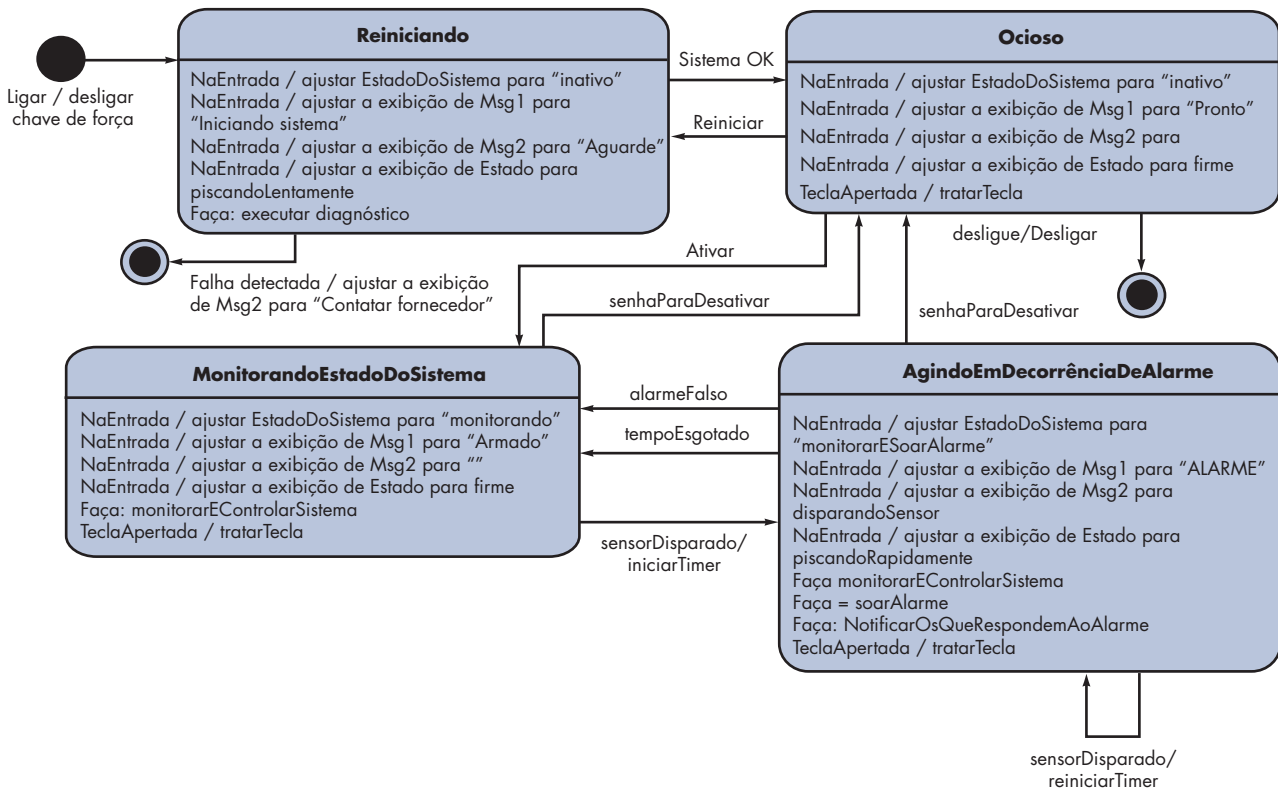
Por exemplo, o diagrama de estados (Figura 7.4) indica que as transições do estado **Ocioso** podem ocorrer se o sistema for reinicializado, ativado ou desligado. Se o sistema for ativado (isto é, o sistema de alarme for ligado), a transição para o estado **MonitorandoEstadoDoSistema** ocorre, as mensagens de tela são alteradas como mostrado e o processo *monitorarEControlarSistema* é chamado. Ocorrem duas transições do estado **MonitorandoEstadoDoSistema** — (1) quando o sistema é desativado, ocorre uma transição de volta para o estado **Ocioso**; (2)

³ Outras notações referentes à modelagem comportamental são apresentadas na Seção 7.3.

⁴ A notação de diagramas de estados aqui usada está em conformidade com a notação da UML. Um “diagrama de transição de estados” se encontra disponível na análise estruturada, porém, o formato UML é superior em termos de representação e conteúdo de informações.

FIGURA 7.4

Diagrama de estados para a função de segurança do CasaSegura



quando um sensor é ativado, uma transição para o estado **AgindoEmDecorrenciaDeAlarme**. Todas as transições e o conteúdo de todos os estados são considerados durante a revisão.

Um modo ligeiramente distinto de representação comportamental é a tabela de ativação de processos (*process activation table*, PAT). A PAT representa informações contidas em diagramas de estados no contexto de processos, e não de estados. A tabela indica quais processos (bolhas) no modelo de fluxos serão chamados quando um evento ocorrer. A PAT pode ser usada como guia para um projetista que tem de construir um executável que controla os processos representados nesse nível. Uma PAT para o modelo de fluxos nível 1 do software *CasaSegura* é mostrada na Figura 7.5.

A CSPEC descreve o comportamento do sistema, mas não nos dá nenhuma informação sobre o funcionamento interno dos processos ativados como resultado desse comportamento. A notação de modelagem que fornece essas informações é discutida na Seção 7.2.4.

7.2.4 A especificação de processos

A *especificação de processos* (*process specification*, PSPEC) é usada para descrever todos os processos do modelo de fluxo que aparecem no nível final de refinamento. O conteúdo da especificação de processos pode incluir texto narrativo, uma descrição⁵ em PDL (linguagem de projeto de programas) do algoritmo de processos, equações matemáticas, tabelas ou diagramas

⁵ A linguagem de projeto de programas (PDL) mistura sintaxe de linguagem de programação com texto narrativo para oferecer um projeto procedural detalhado. A PDL é discutida rapidamente no Capítulo 10.

FIGURA 7.5

Tabela de ativação de processos para a função de segurança do CasaSegura

<u>eventos de entrada</u>						
evento de sensor	0	0	0	0	1	0
flag piscante	0	0	1	1	0	0
iniciar/parar chave	0	1	0	0	0	0
exibir status de ação completa	0	0	0	1	0	0
em andamento	0	0	1	0	0	0
tempo esgotado	0	0	0	0	0	1
<u>saída</u>						
signal de alarme	0	0	0	0	1	0
<u>ativação de processos</u>						
monitorar e controlar sistema	0	1	0	0	1	1
ativar/desativar sistema	0	1	0	0	0	0
exibir mensagens e estado	1	0	1	1	1	1
interagir com o usuário	1	0	0	1	0	1

CASASEGURA



Modelagem de fluxo de dados

Cena: Sala do Jamie, após a última reunião para levantamento de requisitos.

Atores: Jamie, Vinod e Ed — todos membros da equipe de engenharia de software do CasaSegura.

Conversa:

(Jamie esboçou os modelos apresentados nas Figuras 7.1 a 7.5 e está mostrando-os para Ed e Vinod.)

Jamie: Tive um curso de engenharia de software na faculdade e eles nos ensinaram essa matéria. O professor disse que é um tanto antigo, mas, quer saber, ajuda-me a esclarecer certos pontos.

Ed: Isso é excelente. Mas não estou vendo nenhuma classe ou objeto aí.

Jamie: Não... Trata-se apenas de um modelo de fluxos com algo sobre seu comportamento no meio dele.

Vinod: Então estes DFDs representam uma visão E-P-S do software, não é mesmo?

Ed: E-P-S?

Vinod: Entrada-processo-saída. Os DFDs são, na verdade, bastante intuitivos... Se você examiná-los por um momento, eles

mostram como os objetos de dados fluem através do sistema e são transformados à medida que fluem.

Ed: Parece que poderíamos converter cada bolha em um componente executável... Pelo menos no nível mais baixo do DFD.

Jamie: É interessante, podemos fazer isso. De fato, existe uma maneira de transformarmos os DFDs em uma arquitetura de projeto.

Ed: Verdade?

Jamie: Isso mesmo, mas primeiro temos que desenvolver um modelo de requisitos completo e, no momento, ele não está.

Vinod: Bem, esta é a primeira etapa, mas também teremos que tratar os elementos baseados em classes, bem como os aspectos comportamentais, embora o diagrama de estados e a PAT já façam uma parte disso.

Ed: Temos muito trabalho pela frente e não muito tempo para fazê-lo. (Doug — o gerente de engenharia de software — entra na sala.)

Doug: Portanto, os próximos dias serão gastos no desenvolvimento do modelo de requisitos, não é mesmo?

Jamie (expressando orgulho): Já começamos.

Doug: Muito bem, temos muito trabalho pela frente e não muito tempo para fazê-lo.

(Os três engenheiros de software se entreolham e sorriem.)

PONTO-CHAVE

PSPEC é uma “miniespecificação” para cada transformação no nível mais baixo de refinamento de um DFD.

de atividades UML. Ao juntarmos uma PSPEC a cada bolha no modelo de fluxos, podemos criar uma “miniespecificação” que sirva como guia para o projeto do componente de software que irá implementar a bolha.

Para ilustrarmos o uso da PSPEC, consideremos a transformação processar senha representada no modelo de fluxos do CasaSegura (Figura 7.2). A PSPEC para essa função poderia adquirir a seguinte forma:

PSPEC: processar senha (no painel de controle). A transformação *processar senha* realiza a validação de senhas no painel de controle para a função de segurança do CasaSegura. *Processar senha* re-

cebe uma senha de quatro dígitos da função *interagir com o usuário*. Primeiro, a senha é comparada com a senha-mestra armazenada no sistema. Se a senha-mestra coincidir, <valid id message = true> é passada para a função *mostrar mensagem e status*. Se a senha-mestra não coincidir, os quatro dígitos são comparados com uma tabela de senhas secundárias (essas poderiam ser atribuídas a convidados e/ou empregados que precisam ter acesso à casa quando o proprietário não está presente). Se a senha coincidir com uma entrada da tabela, <valid id message = true> é passado para a função *mostrar mensagem e status*. Caso não coincida, <valid id message = false> é passado para a função *mostrar mensagem e status*.

Caso sejam necessários mais detalhes sobre o algoritmo nesse estágio, uma representação PDL também poderia ser incluída como parte da PSPEC. Entretanto, muitos acreditam que a versão PDL deva ser postergada até que se inicie o projeto de componentes.

FERRAMENTAS DO SOFTWARE



Análise estruturada

Objetivo: As ferramentas de análise estruturada permitem a um engenheiro de software criar modelos de dados, modelos de fluxos e modelos comportamentais para possibilitar a verificação de consistência e continuidade, bem como fácil edição e extensão. Os modelos criados empregando-se tais ferramentas fornecem ao engenheiro de software uma visão da representação de análise e podem ajudar a eliminar erros antes que se propaguem pelo projeto ou, pior ainda, na própria implementação.

Mecânica: Ferramentas nessa categoria usam um “dicionário de dados” como banco de dados central para a descrição de todos os objetos de dados. Uma vez que as entradas no dicionário tenham sido definidas, os diagramas entidade-relacionamento podem ser criados e as hierarquias de objetos podem ser desenvolvidas. Os recursos dos diagramas de fluxo de dados permitem criar facilmente esse modelo gráfico, bem como fornecem recursos para a criação de PSPECs e CSPECs.

As ferramentas de análise também possibilitam que o engenheiro de software crie modelos comportamentais usando o diagrama de estados como notação efetiva.

Ferramentas representativas:⁶

MacA&D, *WinA&D*, desenvolvidas pela Excel software (www.excelsoftware.com), oferece um conjunto de ferramentas simples e baratas para análise e projeto tanto para máquinas Macs quanto Windows.

MetaCASE Workbench, desenvolvida pela MetaCase Consulting (www.metacase.com), é uma metaferramenta usada para definir um método de análise ou projeto (inclusive análise estruturada), bem como seus conceitos, regras, notações e geradores.

System Architect, desenvolvida pela Popkin Software (www.popkin.com) oferece uma ampla gama de ferramentas de análise e projeto, inclusive ferramentas para modelagem de dados e análise estruturada.

7.3 CRIAÇÃO DE UM MODELO COMPORTAMENTAL

Como modelar a reação do software a algum evento externo?

A notação de modelagem discutida aqui até então representa elementos estáticos do modelo de requisitos. É chegado o momento de fazermos uma transição para o comportamento dinâmico do sistema ou produto. Para tanto, precisamos representar o comportamento do sistema em função do tempo e eventos específicos.

O *modelo comportamental* indica como o software irá responder a estímulos ou eventos externos. Para criá-lo, devemos executar as seguintes etapas:

1. Avaliar todos os casos de uso para entender completamente a sequência de interação dentro do sistema.
2. Identificar eventos que dirigem a sequência de interações e compreender como esses eventos se relacionam com objetos específicos.
3. Criar uma sequência para cada caso de uso.
4. Construir um diagrama de estados para o sistema.
5. Revisar o modelo comportamental para verificar precisão e consistência.

Cada uma dessas etapas é discutida nas seções a seguir.

⁶ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

7.3.1 Identificação de eventos com o caso de uso

No Capítulo 6 vimos que o caso de uso representa uma sequência de atividades que envolve atores e o sistema. Em geral, um evento ocorre toda vez que o sistema e um ator trocam informações. Na Seção 7.2.3, indicamos que um evento *não* é a informação que foi trocada, mas sim o fato de que as informações foram trocadas.

Um caso de uso é examinado para encontrar pontos de troca de informação. Para ilustrarmos, reconsideremos o caso de uso de uma pequena parte da função de segurança do *CasaSegura*.

O proprietário usa o teclado numérico para digitar uma senha de quatro dígitos. A senha é comparada com a senha válida armazenada no sistema. Se a senha for incorreta, o painel de controle emitirá um bipe e se reiniciará para receber novas entradas. Se a senha for correta, o painel de controle aguarda as próximas ações.

Os trechos sublinhados do cenário do caso de uso indicam eventos. Deve-se identificar um ator para cada evento; as informações trocadas devem ser indicadas e quaisquer condições ou restrições devem ser enumeradas.

Como um exemplo típico de evento, consideremos o trecho sublinhado do caso de uso “proprietário usa o teclado numérico para digitar uma senha de quatro dígitos”. No contexto do modelo de requisitos, o objeto, **Proprietário**,⁷ transmite um evento para o objeto **PainelDeControle**. O evento poderia ser chamado *entrada de senha*. As informações transferidas são os quatro dígitos que constituem a senha, mas essa não é parte essencial do modelo comportamental. É importante notar que alguns eventos têm um impacto explícito no fluxo de controle do caso de uso, ao passo que outros não têm impacto direto no fluxo de controle. Por exemplo, o evento *entrada de senha* não muda explicitamente o fluxo de controle do caso de uso, mas os resultados do evento *entrada de senha* (derivado da interação “senha é comparada com a senha válida armazenada no sistema”) terá um impacto explícito no fluxo de controle e informações do software *CasaSegura*.

Uma vez que todos os eventos tenham sido identificados, são alocados aos objetos envolvidos. Os objetos podem ser responsáveis pela geração de eventos (por exemplo, **Proprietário** gera um evento *entrada de senha*) ou reconhece eventos que ocorreram em algum outro ponto (por exemplo, **PainelDeControle** reconhece o resultado binário do evento *comparação de senha*).

7.3.2 Representações de estados

No contexto da modelagem comportamental, devem ser consideradas duas caracterizações de estados distintas: (1) o estado de cada classe à medida que o sistema executa sua função e (2) o estado do sistema como observado de fora à medida que o sistema executa sua função.⁸

O estado de uma classe pode assumir tanto características passivas quanto ativas [Cha93]. *Estado passivo* é o estado atual de todos os atributos de um objeto. Por exemplo, o estado passivo da classe **Jogador** (na aplicação de videogame discutida no Capítulo 6) incluiria atributos referentes à **posição** e **orientação** atual do **Jogador**, bem como outros recursos do **Jogador** relevantes ao jogo (por exemplo, um atributo que indique **permanência de desejos mágicos**). O *estado ativo* de um objeto indica o estado atual do objeto à medida que passa por uma transformação ou processamento contínuo. A classe **Jogador** poderia ter os seguintes estados ativos: *em movimento*, *em repouso*, *contundido*, *recuperando-se no departamento médico*; *preso*, *perdido* e assim por diante. Deve acontecer um evento (algumas vezes denominado *gatilho*). Para forçar um objeto a fazer uma transição de um estado ativo para outro.

PONTO-CHAVE

O sistema possui estados que representam comportamento específico externamente observável; uma classe possui estados que representam seu comportamento à medida que o sistema executa suas funções.

⁷ Neste exemplo, partimos do pressuposto de que cada usuário (proprietário do imóvel) que interage com o *CasaSegura* possui uma senha de identificação e é, consequentemente, um objeto legítimo.

⁸ Os diagramas de estados apresentados no Capítulo 6 e na Seção 7.3.2 representam o estado do sistema. Nossa discussão nessa seção irá se concentrar no estado de cada classe do modelo de análise.

Nos próximos parágrafos serão discutidas duas representações comportamentais distintas. A primeira indica como determinada classe muda de estado baseada em eventos externos e a segunda mostra o comportamento do software em função do tempo.

Diagramas de estados para classes de análise. O componente de um modelo comportamental é um diagrama de estados⁹ UML que representa estados ativos para cada uma das classes e para os eventos (*gatilhos*) que provocam mudanças entre esses estados ativos. A Figura 7.6 ilustra um diagrama de estados para o objeto **PainelDeControle** na função de segurança *CasaSegura*.

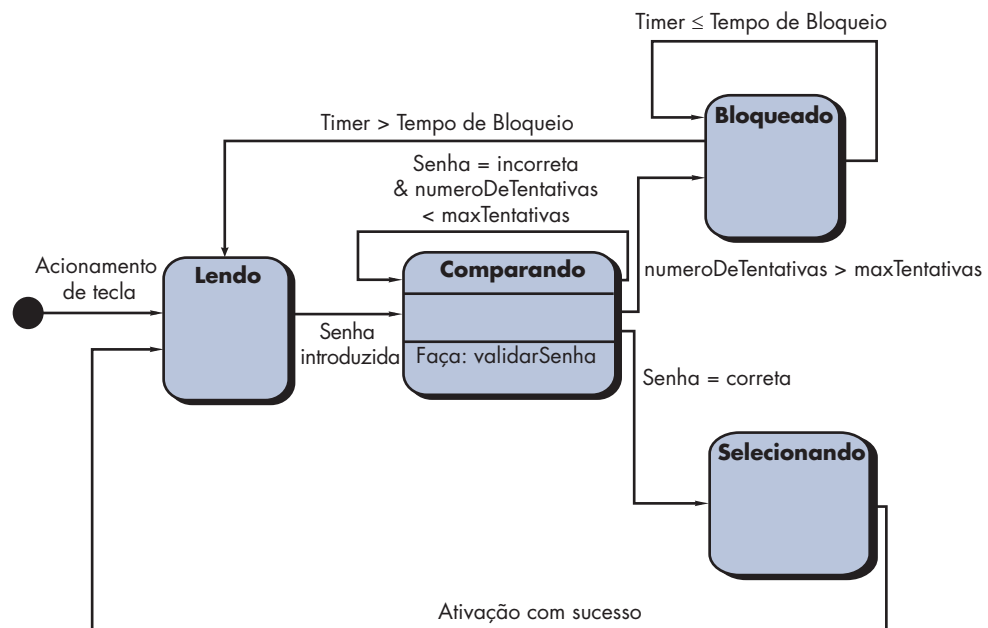
Cada seta da Figura 7.6 representa a transição de um estado ativo de um objeto para outro. As identificações em cada seta representam o evento que dispara a transição. Embora o modelo de estados ativos dê uma visão proveitosa sobre a “história de vida” de um objeto, é possível especificar informações adicionais para uma maior profundidade no entendimento do comportamento de um objeto. Além de especificarmos o evento que faz com que a transição ocorra, podemos especificar um guarda e uma ação [Cha93]. *Guarda* é uma condição booleana que deve ser satisfeita de modo que a transição ocorra. Por exemplo, o guarda para a transição do estado “lendo” para o estado “comparando” na Figura 7.6 pode ser determinado examinando-se o caso de uso:

if (entrada de senha = 4 dígitos) então compare com a senha armazenada

Em geral, o guarda para uma transição normalmente depende do valor de um ou mais atributos de um objeto. Em outras palavras, o guarda depende do estado passivo do objeto.

Uma *ação* ocorre concorrentemente com a transição de estado ou como consequência dele e geralmente envolve uma ou mais operações (responsabilidades) do objeto. Por exemplo, a ação ligada ao evento *entrada de senha* (Figura 7.6) é uma operação chamada *validarSenha()* que acessa um objeto **senha** e realiza uma comparação dígito por dígito para validar a senha introduzida.

FIGURA 7.6
Diagrama de estados
para a classe
PainelDeControle



⁹ Caso não esteja familiarizado com a UML, apresentamos uma breve introdução a essa importante notação de modelagem no Apêndice 1.

Diagramas de sequência. O segundo tipo de representação comportamental, denominado *diagrama de sequência* em UML, indica como os eventos provocam transições de objeto para objeto. Uma vez que os eventos tenham sido identificados pelo exame de um caso de uso, o modelador cria um diagrama de sequência — uma representação de como os eventos provocam o fluxo de um objeto para outro em função do tempo. Em resumo, o diagrama de sequência é uma versão abreviada do caso de uso. Ele representa classes-chave e os eventos que fazem com que o comportamento flua de classe para classe.

A Figura 7.7 ilustra um diagrama de sequência parcial para a função de segurança do *CasaSegura*. Cada uma das setas representa um evento (derivado de um caso de uso) e indica como o evento canaliza o comportamento entre os objetos do *CasaSegura*. O tempo é medido verticalmente (de cima para baixo), e os retângulos estreitos na vertical representam o tempo gasto no processamento de uma atividade. Os estados poderiam ser mostrados ao longo de uma linha do tempo vertical.

O primeiro evento, *sistema pronto*, é derivado do ambiente externo e canaliza comportamento para o objeto **Proprietário**. O proprietário do imóvel introduz uma senha. Um evento *de solicitação de busca* é passado para **Sistema**, que busca uma senha em um banco de dados simples e retorna um *resultado* (*encontrada* ou *não encontrada*) para **PainelDeControle** (agora no estado *comparando*). Uma senha válida resulta em um evento *senha=correta* para **Sistema**, que ativa **Sensores** com um evento *solicitação de ativação*. Por fim, o controle retorna ao proprietário com o evento *ativação com sucesso*.

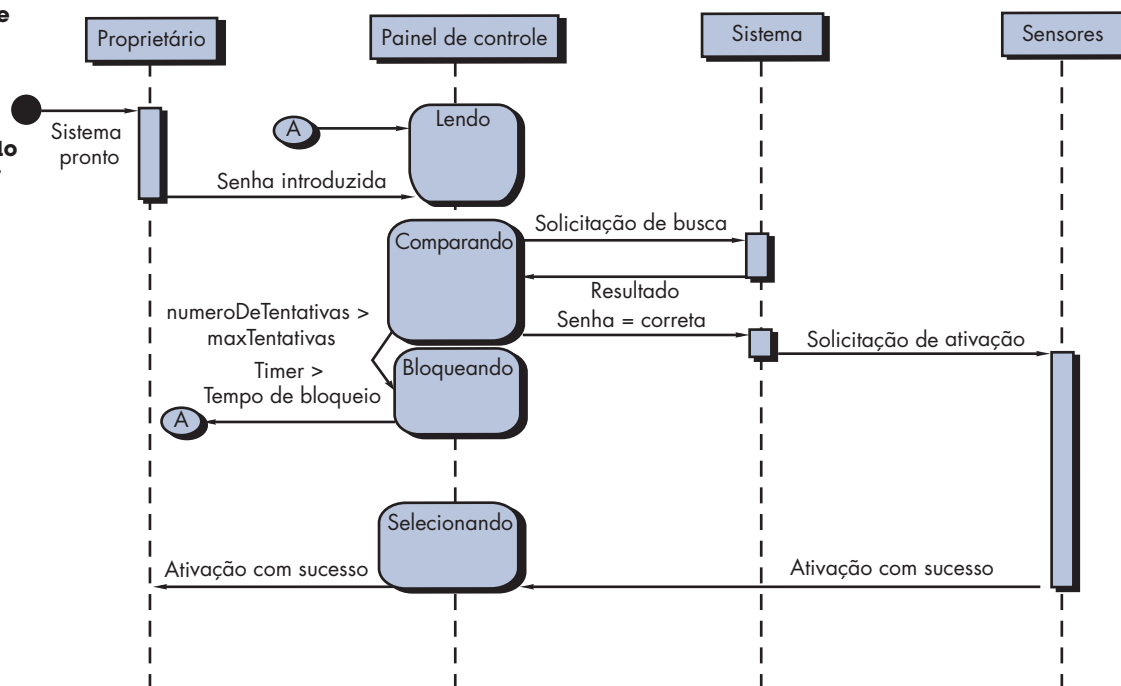
Assim que um diagrama de sequência completo tiver sido desenvolvido, todos os eventos que provocam transições entre objetos do sistema podem ser reunidos em um conjunto de eventos de entrada e eventos de saída (de um objeto). Essas informações são úteis na criação de um projeto efetivo para o sistema a ser construído.

PONTO-CHAVE

Diferentemente de um diagrama de estados que representa comportamento sem citar as classes envolvidas, um diagrama de sequência representa comportamento descrevendo como as classes passam de um estado para outro.

FIGURA 7.7

Diagrama de sequência (parcial) para a função de segurança do CasaSegura



FERRAMENTAS DO SOFTWARE



Modelagem de análise generalizada em UML

Objetivo: As ferramentas de modelagem de análise fornecem a capacidade de desenvolver modelos baseados em cenários, modelos baseados em classes e modelos comportamentais usando a notação UML.

Mecânica: As ferramentas nesta categoria suportam todo o conjunto de diagramas UML necessários para construir um modelo de análise (as ferramentas também dão suporte à modelagem de projeto). Além da diagramação, elas (1) realizam verificação de consistência e correção para todos os diagramas UML, (2) fornecem links para projeto e geração de código, (3) constroem um banco de dados que permite o gerenciamento e a avaliação de grandes modelos UML necessários para sistemas complexos.

Ferramentas representativas:¹⁰

As seguintes ferramentas suportam todo o conjunto de diagramas UML necessário para a modelagem de análise:

ArgoUML é uma ferramenta com código-fonte aberto disponível em argouml.tigris.org.

Enterprise Architect, desenvolvida pela Sparx Systems (www.sparxsystems.com.au).

PowerDesigner, desenvolvida pela Sybase (www.sybase.com).

Rational Rose, desenvolvida pela IBM (Rational) (www01.ibm.com/software/rational/).

System Architect, desenvolvida pela Popkin Software (www.popkin.com).

UML Studio, desenvolvida pela Pragsoft Corporation (www.pragsoft.com).

Visio, desenvolvida pela Microsoft (www.microsoft.com).

Visual UML, desenvolvida pela Visual Object Modelers (www.visualuml.com).

7.4 PADRÕES PARA A MODELAGEM DE REQUISITOS

Padrões de software constituem um mecanismo para capturar conhecimento do domínio acumulado para permitir que seja reaplicado quando um novo problema é encontrado. Em alguns casos, o conhecimento do domínio é aplicado a um novo problema no mesmo domínio de aplicação. Em outros casos, o conhecimento do domínio capturado por um padrão pode ser aplicado por analogia a um domínio de aplicação completamente diferente.

O autor original de um padrão de análise não “cria” o padrão, mas sim, o *descobre* à medida que o fluxo de trabalho de levantamento de requisitos é conduzido. Assim que o padrão tiver sido descoberto, é documentado descrevendo-se “explicitamente o problema geral ao qual o padrão se aplica, a solução recomendada, hipóteses e restrições do emprego do padrão na prática e em geral algumas outras informações sobre o padrão, como a motivação e as forças que orientam o uso do padrão, discussão das vantagens e desvantagens do padrão, bem como referências para alguns exemplos conhecidos do uso desse padrão em aplicações práticas”. [Dev01].

No Capítulo 5, introduzimos o conceito de padrões de análise e indicamos que padrões representam uma solução que frequentemente incorpora uma classe, função ou comportamento em um campo de aplicação. O padrão pode ser reutilizado ao se realizar a modelagem de requisitos para uma aplicação em um domínio.¹¹ Os padrões de análise são armazenados em um repositório de modo que os membros da equipe de software possam usar recursos de pesquisa para encontrá-los e reutilizá-los. Assim que um padrão apropriado tiver sido selecionado, é integrado ao modelo de requisitos por meio de referência ao nome do padrão.

7.4.1 Descoberta de padrões de análise

O modelo de requisitos é formado por uma ampla gama de elementos: baseados em cenários (casos de uso), orientados a dados (o modelo de dados), baseados em classes, orientados a fluxos e comportamentais. Cada um deles examina o problema segundo uma perspectiva e cada uma delas proporciona a descoberta de padrões que poderiam ocorrer em um domínio de aplicação, ou por analogia, em domínios de aplicação diferentes.

¹⁰ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

¹¹ Uma discussão aprofundada do uso de padrões durante o projeto de software é apresentada no Capítulo 12.

O elemento mais básico na descrição de um modelo de requisitos é o caso de uso. No contexto desta discussão, um conjunto coerente de casos de uso poderia servir como base para descobrir um ou mais padrões de análise. *Padrão de análise semântica* (*semantic analysis pattern, SAP*) “descreve um pequeno conjunto de casos de uso coerentes que juntos descrevem uma aplicação genérica básica” [Fer00].

Consideremos o seguinte caso de uso preliminar para um software necessário para controlar e monitorar um sensor de proximidade e câmera de visualização real para um automóvel:

Caso de uso: Monitorar deslocamento em marcha a ré

Descrição: Quando o veículo é colocado em *marcha a ré*, o software de controle habilita um alimentador de vídeo por meio de uma câmera de vídeo colocada atrás do painel de comandos. O software de controle sobrepõe uma variedade de linhas de orientação e distâncias na exibição do painel de comandos de modo que o condutor do veículo possa ser orientado à medida que se desloca em marcha a ré. O software de controle também monitora um sensor de proximidade para determinar se um objeto se encontra ou não a 3 m da traseira do veículo. Ele irá frear o veículo automaticamente se o sensor de proximidade indicar um objeto a x metros da traseira do veículo, em que x é determinado com base na velocidade do veículo.

Esse caso de uso implica uma funcionalidade muito variada que poderia ser refinada e elaborada (em um conjunto de casos de uso coerentes) durante o levantamento de requisitos e a modelagem. Independentemente do nível de elaboração alcançado, os casos de uso sugerem um SAP simples, porém de larga aplicação — o monitoramento e controle de sensores e atuadores baseado em software em um sistema físico. Nesse caso, os “sensores” fornecem informações sobre proximidade e informações de vídeo. O “atuador” é o sistema de frenagem do veículo (acionado caso um objeto esteja muito próximo do veículo). Porém, em um caso mais genérico, se descobre um padrão de larga aplicação.

Diferentes domínios de aplicação para monitorar sensores e controlar atuadores físicos necessitam de software. Um padrão de análise que descreva requisitos genéricos para essa capacidade poderia ser utilizado largamente. O padrão, denominado **Atuador-Sensor**, seria aplicável como parte do modelo de requisitos do *CasaSegura* e é discutido na Seção 7.4.2, a seguir.

7.4.2 Exemplo de padrão de requisitos: atuador-sensor¹²

Um dos requisitos da função de segurança do *CasaSegura* é a habilidade de monitorar sensores de segurança (por exemplo, sensores contra roubos, sensores de fogo, fumaça ou CO, sensores de água).

Extensões para o *CasaSegura* baseadas na Internet exigiram a capacidade de controlar o movimento (por exemplo, deslocamento horizontal e vertical, ampliação/redução) de uma câmera de segurança no interior de uma residência. A implicação — o software *CasaSegura* deve gerenciar vários sensores e “atuadores” (por exemplo, mecanismos de controle de câmera).

Konrad e Cheng [Kon02] sugeriram um padrão de requisitos chamado **Atuador-Sensor** que fornece uma útil orientação para modelar esses requisitos no software *CasaSegura*. Uma versão abreviada do padrão **Atuador-Sensor**, originalmente desenvolvido para aplicações na indústria automobilística, é apresentada a seguir.

Nome do Padrão. Atuador-Sensor

Intuito. Especificar vários tipos de sensores e atuadores em um sistema embutido.

Motivo. Os sistemas embarcados normalmente possuem vários tipos de sensores e atuadores. Esses sensores e atuadores estão todos direta ou indiretamente conectados a uma unidade de controle. Embora muitos sensores e atuadores pareçam bem diferentes, seu comportamento é suficientemente similar para estruturá-los em um padrão. O padrão mostra como especificar os sensores e atuadores para um sistema, inclusive atributos e operações. O padrão **Atuador-Sensor** usa um mecanismo *pull* (solicitação explícita de informação)

12 Essa seção foi adaptada de [Kon02] com a permissão dos autores.

para **PassiveSensors (SensoresPassivos)** e um mecanismo *push* (difusão de informação) para os **ActiveSensors (SensoresAtivos)**.

Restrições

- Cada sensor passivo deve ter algum método para ler entrada de sensores e atributos que representam o valor do sensor.
- Cada sensor ativo deve ter capacidades para transmitir mensagens de atualização quando seu valor muda.
- Cada sensor ativo deve enviar um *senal de vida*, uma mensagem de estado emitida em determinado intervalo de tempo, para detectar defeitos.
- Cada atuador deve ter algum método para chamar a resposta apropriada determinada por **ComputingComponent (ComponenteDeCálculo)**.
- Cada sensor e atuador deve ter uma função implementada para verificar seu próprio estado de operação.
- Cada sensor e atuador deve ser capaz de testar a validade dos valores recebidos ou enviados e estabelecer seu estado de operação caso os valores se encontrem fora das especificações.

Aplicabilidade. Útil em qualquer sistema em que vários sensores e atuadores estão presentes.

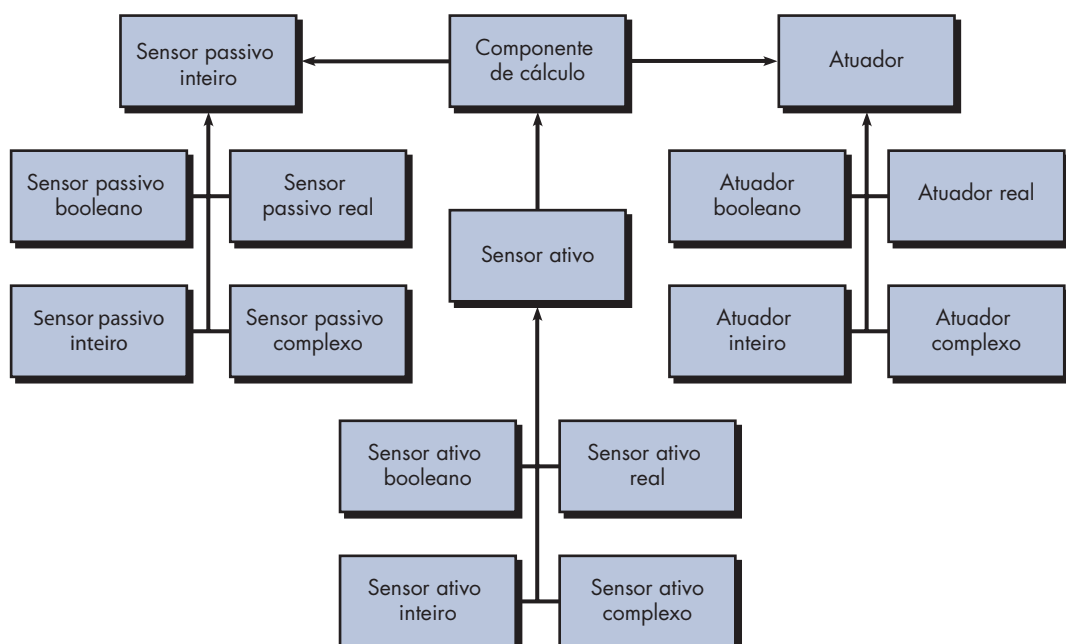
Estrutura. Um diagrama de classes UML para o padrão **Atuador-Sensor** é mostrado na Figura 7.8. **Atuador**, **PassiveSensor** e **ActiveSensor** são classes abstratas e grafadas em itálico. Existem quatro tipos diferentes de sensores e atuadores nesse padrão.

As classes **Boolean (Booleana)**, **Integer (Inteira)** e **Real (Real)** representam os tipos mais comuns de sensores e atuadores. As classes complexas são sensores ou atuadores que usam valores que não podem ser facilmente representados em termos de tipos de dados primitivos, como um dispositivo de radar. Não obstante, tais dispositivos ainda deveriam herdar a interface das classes abstratas já que elas deveriam ter funcionalidades básicas como consulta aos estados de operação.

FIGURA 7.8

Diagrama de sequência UML para o padrão Atuador-Sensor.

Fonte: adaptado de [Kon02] com permissão



Comportamento. A Figura 7.9 apresenta um diagrama de sequência UML para um exemplo do padrão **Atuador-Sensor** já que ele poderia ser aplicado à função do *CasaSegura* que controla o posicionamento (por exemplo, deslocamentos, ampliação/redução) de uma câmera de segurança. Nesse caso, **ControlPanel (PainelControle)**¹³ consulta um sensor (um sensor de posição passivo) e um atuador (controle de deslocamentos) para verificar o estado de operação para fins de diagnóstico antes de ler ou configurar um valor. As mensagens *Set Physical Value (AjustarValorFísico)* e *Get Physical Value (ObterValorFísico)* não são mensagens entre objetos. Em vez disso, descrevem a interação entre os dispositivos físicos do sistema e seus equivalentes em software. Na parte inferior do diagrama, abaixo da linha horizontal, **PositionSensor (SensorPosicional)** informa que o estado de operação é zero. **ComputingComponent** (representada por **ControlPanel**) envia então o código de erro devido a uma falha no sensor de posição para **FaultHandler (ControleFalhas)** que decidirá como esse erro afeta o sistema e que medidas são necessárias. Ela obtém os dados dos sensores e calcula a resposta necessária para os atuadores.

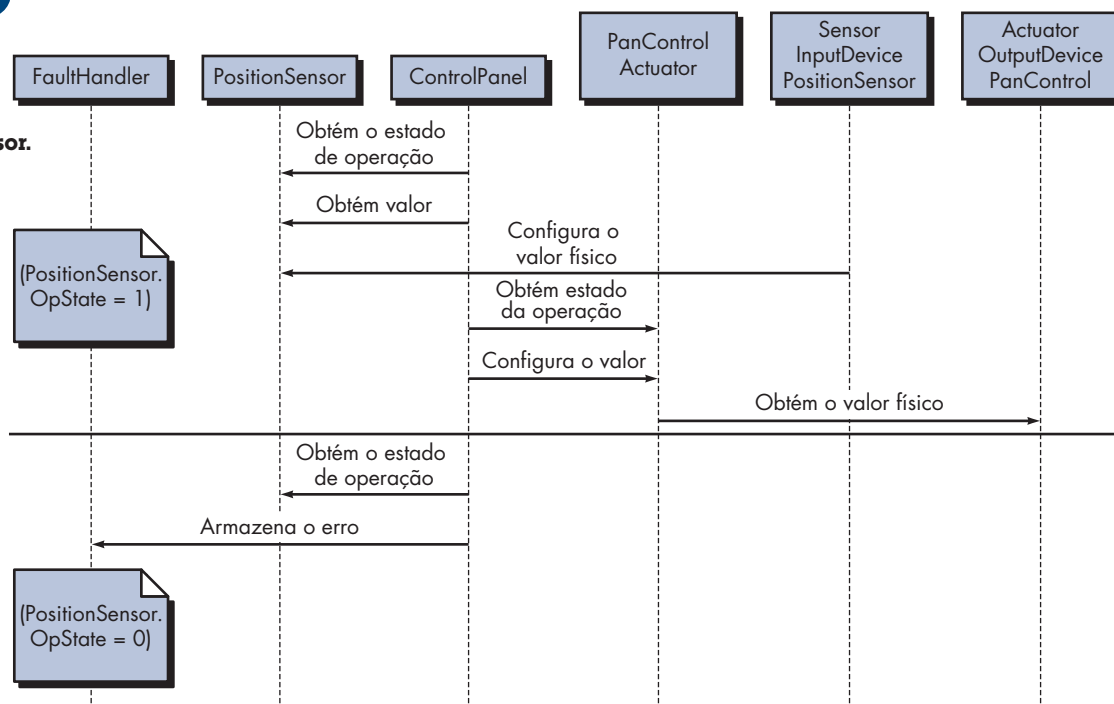
Participantes. Essa seção de descrição dos padrões “elencas as classes/objetos incluídos no padrão de requisitos” [Kon02] e descreve as responsabilidades de cada classe/objeto (Figura 7.8). A seguir, temos uma lista resumida:

- **PassiveSensor abstract:** Define uma interface para sensores passivos.
- **PassiveBooleanSensor:** Define sensores passivos booleanos.
- **PassiveIntegerSensor:** Define sensores passivos inteiros.
- **PassiveRealSensor:** Define sensores passivos reais.
- **ActiveSensor abstract:** Define uma interface para sensores ativos.
- **ActiveBooleanSensor:** Define sensores ativos booleanos.
- **ActiveIntegerSensor:** Define sensores ativos inteiros.

FIGURA 7.9

Diagrama de classes UML para o padrão **Atuador-Sensor**.

Fonte: reimpresso de [Kon02] com permissão



13 O padrão original usa o termo genérico **ComputingComponent**.

- **ActiveRealSensor:** Define sensores ativos reais.
- **Actuator abstract:** Define uma interface para atuadores.
- **BooleanActuator:** Define atuadores booleanos.
- **IntegerActuator:** Define atuadores inteiros.
- **RealActuator:** Define atuadores reais.
- **ComputingComponent:** A parte principal do controlador; ele obtém os dados dos sensores e calcula a resposta necessária para os atuadores.
- **ActiveComplexSensor:** Os sensores ativos complexos possuem a funcionalidade básica da classe abstrata **ActiveSensor**, porém, métodos e atributos adicionais mais elaborados precisam ser especificados.
- **PassiveComplexSensor:** Os sensores passivos complexos possuem a funcionalidade básica da classe abstrata **PassiveSensor**, porém, métodos e atributos adicionais mais elaborados precisam ser especificados.
- **ComplexActuator:** Os atuadores complexos também possuem a funcionalidade básica da classe abstrata **Actuator**, porém, métodos e atributos adicionais mais elaborados precisam ser especificados.

Colaborações. Essa seção descreve como os objetos e as classes interagem entre si e como cada uma delas cumpre suas obrigações.

- Quando **ComputingComponent** precisar atualizar o valor de um **PassiveSensor**, ela consulta os sensores, solicitando o valor por meio do envio de uma mensagem apropriada.
- **ActiveSensors** não são consultados. Eles iniciam a transmissão de valores de sensor para a unidade de cálculo, usando o método apropriado para configurar o valor no **ComputingComponent**. Eles enviam um sinal de vida pelo menos uma vez durante um intervalo de tempo especificado para atualizar seus registros de horas com o horário do relógio do sistema.
- Quando **ComputingComponent** precisar configurar o valor de um atuador, ela envia o valor para o atuador.
- **ComputingComponent** pode consultar e configurar o estado de operação dos sensores e atuadores usando os métodos apropriados. Se for constatado que um estado de operação é zero, o erro é enviado a **FaultHandler**, uma classe que contém métodos para tratar mensagens de erro como, por exemplo, acionar um mecanismo de recuperação mais elaborado ou um dispositivo de backup. Se a recuperação não for possível, o sistema poderá usar apenas o último valor conhecido do sensor ou o valor default.
- **ActiveSensors** oferece métodos para acrescentar ou remover os endereços ou intervalos de endereços dos componentes que querem receber as mensagens no caso de mudança de um valor.

Consequências

1. As classes de sensores e atuadores possuem uma interface comum.
2. Os atributos de classes podem ser acessados apenas através de mensagens e a classe decide se aceita ou não uma mensagem. Por exemplo, se o valor de um atuador estiver configurado acima de seu valor máximo, a classe do atuador talvez não aceite a mensagem ou então use um valor máximo padrão.
3. A complexidade do sistema é potencialmente reduzida devido à uniformidade das interfaces para atuadores e sensores.

A descrição dos padrões de requisitos também poderia fornecer referências a outros padrões de projeto e de requisitos relacionados.

7.5 MODELAGEM DE REQUISITOS PARA WEBAPPS¹⁴

Os desenvolvedores Web normalmente são céticos quando lhes é sugerida a ideia de análise de requisitos para WebApps. “Afim de contas”, argumentam eles, “o processo de desenvolvimento para Web deve ser ágil e a análise toma muito tempo. Ela irá nos retardar quando simplesmente precisamos projetar e construir uma WebApp.”

O levantamento de requisitos realmente toma tempo, porém resolver o problema errado leva mais tempo ainda. A pergunta para cada desenvolvedor de WebApp é simples — você tem certeza de que entendeu os requisitos do problema? Se a resposta for um inequívoco “sim”, poderia ser possível pular a fase de modelagem de requisitos, porém se a resposta for “não”, a modelagem de requisitos deve ser realizada.

7.5.1 Que nível de análise é suficiente?

O grau com o qual a modelagem de requisitos para WebApps é enfatizado depende dos seguintes fatores:

- Tamanho e complexidade do incremento de WebApp.
- Número de interessados (a análise pode ajudar a identificar requisitos conflitantes provenientes de várias fontes).
- Tamanho da equipe de desenvolvimento de WebApps.
- Nível em que os membros da equipe de desenvolvimento de WebApps trabalharam juntos antes (a análise pode ajudar a desenvolver um entendimento comum do projeto).
- Nível de êxito da organização é diretamente dependente do êxito das WebApps.

O contrário dos pontos anteriores é que à medida que um projeto se torna menor, que o número de interessados diminui, que a equipe se torne mais coesa e a aplicação seja menos crítica, é razoável aplicar uma abordagem de análise menos rígida.

Embora seja uma boa ideia analisar o problema *antes* de começar o projeto, não é verdade que *toda* análise deve preceder *todo* o projeto. De fato, o projeto de uma parte específica da WebApp exige apenas uma análise daqueles requisitos que afetam apenas essa parte da WebApp. Um exemplo para o *CasaSegura*: poderíamos projetar de forma válida toda a estética do site (layouts, esquemas de cores etc.) sem ter analisado as necessidades funcionais para recursos de comércio eletrônico. Precisaríamos analisar apenas aquela parte do problema relevante ao trabalho de projeto do incremento a ser entregue.

7.5.2 Entrada da modelagem de requisitos

Uma versão ágil do processo de software genérico discutido no Capítulo 2 pode ser aplicada quando WebApps forem criadas. O processo incorpora uma atividade de comunicação que identifica interessados e categorias de usuário, o contexto de negócio, metas de aplicação e de informação definidas, requisitos gerais da WebApp e cenários de uso — as informações se tornam entrada para a modelagem de requisitos. Estas são representadas na forma de descrições de linguagem natural, descrições gerais, esboços e outras representações de informação.

A análise captura essas informações, as estruturam usando um esquema de representação formalmente definido (onde apropriado) e depois produz modelos mais rigorosos como saída. O modelo de requisitos fornece uma indicação detalhada da verdadeira estrutura do problema e dá uma visão da forma da solução.

A função **ACS-EVC** (vigilância através de câmeras) do *CasaSegura* foi introduzida no Capítulo 6. Quando foi introduzida, essa função parecia relativamente clara e foi descrita com certo nível de detalhamento como parte de um caso de uso (Seção 6.2.1). Entretanto, um

¹⁴ Essa seção foi adaptada de Pressman e Lowe [Pre08] com permissão.

reexame do caso de uso poderia revelar informações que estão faltando, ambíguas ou não claras. Alguns aspectos de informações faltantes emergiriam naturalmente durante o projeto. Exemplos poderiam incluir o layout específico dos botões de função, seu aspecto estético, o tamanho das visões instantâneas, a colocação de visões das câmeras e planta do imóvel ou até mesmo minúcias como o comprimento máximo e mínimo das senhas. Alguns desses aspectos são decisões de projeto (como o layout dos botões) e outros são requisitos (como o tamanho das senhas) que, fundamentalmente, não influenciam os trabalhos iniciais de projeto.

Porém, algumas informações faltantes poderiam realmente influenciar o próprio projeto como um todo e estarem mais relacionadas a um real entendimento dos requisitos. Por exemplo:

- P1: Qual a resolução de vídeo fornecida pelas câmeras do *CasaSegura*?
- P2: O que acontece se uma condição de alarme for encontrada enquanto a câmera estiver sendo monitorada?
- P3: Como o sistema manipula câmeras que possam ser deslocadas e ampliadas/reduzidas?
- P4: Que informações deveriam ser fornecidas juntamente com a visão das câmeras? (Por exemplo, posição? hora/data? último acesso anterior?)

Nenhuma dessas perguntas foram identificadas ou consideradas no desenvolvimento inicial do caso de uso, muito embora as respostas poderiam ter um efeito substancial em diferentes aspectos do projeto.

Consequentemente, é razoável concluir que embora a atividade de comunicação forneça uma boa fundamentação para o entendimento, a análise de requisitos refina esse entendimento através de interpretações adicionais. À medida que a estrutura do problema é delineada como parte do modelo de requisitos, invariavelmente surgem perguntas. São essas perguntas que preenchem as lacunas — ou, em alguns casos, realmente nos ajudam, em primeiro lugar, a encontrar as lacunas.

Em suma, as entradas para o modelo de requisitos serão as informações coletadas durante a atividade de comunicação — qualquer uma, desde um e-mail informal até uma descrição de projeto detalhado contendo cenários de uso e especificações de produto completas.

7.5.3 Saída da modelagem de requisitos

A análise de requisitos fornece um mecanismo disciplinado para representar e avaliar o conteúdo e função de uma WebApp, os modos de interação que os usuários irão encontrar e o ambiente e a infraestrutura em que a WebApp reside.

Cada uma dessas características pode ser representada como um conjunto de modelos que permitem que os requisitos da WebApp sejam analisados de maneira estruturada. Embora os modelos específicos dependam em grande parte da natureza da WebApp, há cinco classes principais de modelos:

- **Modelo de conteúdo** — identifica o espectro completo do conteúdo a ser fornecido pela WebApp. Conteúdo pode ser texto, gráficos, imagens, vídeo e áudio.
- **Modelo de interações** — descreve a maneira através da qual os usuários interagem com a WebApp.
- **Modelo funcional** — define as operações que serão aplicadas ao conteúdo da WebApp e descreve outras funções de processamento independentes do conteúdo, mas necessárias para o usuário final.
- **Modelo de navegação** — define a estratégia geral de navegação para a WebApp.
- **Modelo de configuração** — descreve o ambiente e a infraestrutura na qual a WebApp reside.

Podemos desenvolver cada um desses modelos usando um esquema de representação (em geral chamado “linguagem”) que permite que seu intuito e estrutura sejam comunicados e avaliados facilmente entre os membros da equipe de engenharia de Web e outros interessados. Como consequência, várias questões fundamentais (por exemplo, erros, omissões, inconsistências, sugestões para aperfeiçoamento ou modificações, pontos a ser esclarecidos) são identificadas e posteriormente trabalhadas.

7.5.4 Modelo de conteúdo para WebApps

O modelo de conteúdo contém elementos estruturais que dão uma visão importante dos requisitos de conteúdo para uma WebApp. Estes englobam objetos de conteúdo e todas as classes de análise — entidades visíveis aos usuários criadas ou manipuladas à medida que o usuário interage com a WebApp.¹⁵

O conteúdo pode ser desenvolvido antes da implementação da WebApp, enquanto a WebApp está sendo construída ou bem depois de estar em funcionamento. Em todos os casos, ela é incorporada via referência navegacional na estrutura geral da WebApp. Um *objeto de conteúdo* poderia ser uma descrição textual de um produto, um artigo descrevendo um evento de noticiário, uma fotografia de ação tirada durante um evento esportivo, a resposta de um usuário em um fórum de discussão, uma animação do logotipo de uma empresa, um breve vídeo de apresentação ou um áudio agregado a um conjunto de slides de uma apresentação. Os objetos de conteúdo poderiam ser armazenados como arquivos distintos, incorporados diretamente em páginas Web ou obtidos dinamicamente de um banco de dados. Em outras palavras, um objeto de conteúdo é qualquer informação coesa que deve ser apresentada a um usuário final.

Os objetos de conteúdo podem ser determinados diretamente dos casos de uso, examinando-se a descrição do cenário para referências diretas e indiretas ao conteúdo. Por exemplo, uma WebApp que oferecesse suporte ao *CasaSegura* seria estabelecida em **CasaSeguraGarantida.com**. Um caso de uso, *Comprando Componentes Específicos do CasaSegura*, descreve o cenário necessário para adquirir um componente do *CasaSegura* e contém a seguinte sentença:

Serei capaz de obter informações descritivas e de preço para cada componente do produto.

O modelo de conteúdo deve ser capaz de descrever o objeto de conteúdo **Componente**. Em muitos casos, uma simples lista dos objetos de conteúdo, com uma breve descrição de cada objeto, é suficiente para definir os requisitos para o conteúdo a ser projetados e implementados. Entretanto, em alguns casos, o modelo de conteúdo pode se beneficiar de uma análise mais rica que ilustre graficamente os relacionamentos entre os objetos de conteúdo e/ou a hierarquia do conteúdo mantido por uma WebApp.

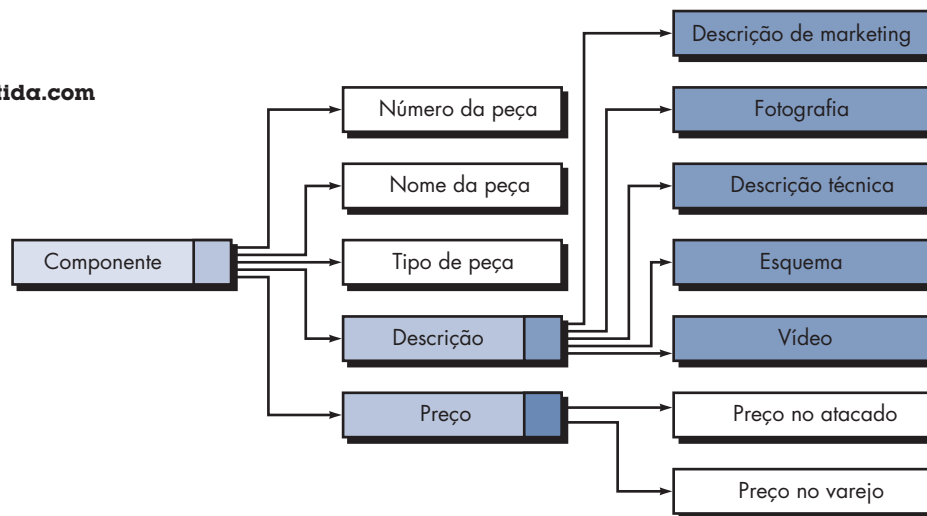
Por exemplo, consideremos a *árvore de dados* [Sri01] criada para um componente de **CasaSeguraGarantida.com** e ilustrada na Figura 7.10. A árvore representa uma hierarquia de informações usadas para descrever um componente. Dados simples ou compostos (um ou mais valores de dados) são representados com retângulos de fundo branco. Os objetos de conteúdo são representados como retângulos chapados. Na figura, a **descrição** é definida por cinco objetos de conteúdo (os retângulos chapados). Em alguns casos, um ou mais desses objetos seriam ainda mais refinados à medida que a árvore de dados fosse se expandindo.

Uma árvore de dados pode ser criada para qualquer conteúdo composto de vários objetos de conteúdo e dados. A árvore de dados é desenvolvida em uma tentativa de definir relações hierárquicas entre os objetos de conteúdo e de fornecer um meio para revisar o conteúdo de modo que omissões e inconsistências sejam reveladas antes de o projeto começar. Além disso, a árvore de dados serve como base para o projeto do conteúdo.

¹⁵ As classes de análise foram discutidas no Capítulo 6.

FIGURA 7.10

Árvore de dados para um componente de CasaSeguraGarantida.com



7.5.5 Modelo de interações para WebApps

A grande maioria das WebApps possibilita um “diálogo” entre o usuário final e a funcionalidade da aplicação, o conteúdo e o comportamento de uma aplicação. Esse diálogo pode ser descrito por meio de um *modelo de interações* que pode ser composto de um ou mais dos seguintes elementos: (1) casos de uso, (2) diagramas de sequência, (3) diagramas de estados,¹⁶ e/ou (4) protótipos de interfaces do usuário.

Em muitos casos, um conjunto de casos de uso já basta para descrever a interação em um nível de análise (um maior refinamento e detalhes serão introduzidos durante a fase de projeto). Entretanto, quando a sequência de interação for complexa e envolver várias classes de análise ou muitas tarefas, às vezes vale a pena representá-la usando uma forma esquemática mais rigorosa.

O layout da interface do usuário, o conteúdo que ela apresenta, os mecanismos de interação que implementa e a estética geral das conexões de WebApps para usuários têm muito a ver com a satisfação do usuário e com o êxito geral da WebApp. Embora possa se argumentar que a criação de um protótipo de interface do usuário seja uma atividade de projeto, é uma boa ideia realizá-la durante a criação do modelo de análise. O quanto antes uma representação física de uma interface do usuário puder ser revisada, maior a probabilidade de que os usuários finais obterão aquilo que realmente desejam. O projeto de interfaces do usuário é discutido em detalhes no Capítulo 11.

Pelo fato de as ferramentas para construção de WebApps serem abundantes, relativamente baratas e funcionalmente poderosas, é melhor criar o protótipo de interface usando-as. O protótipo deve implementar os principais links de navegação e representar o layout geral da tela em grande parte da mesma forma que será construído. Por exemplo, se o intuito é oferecer cinco funções principais de sistema para o usuário final, o protótipo deve representá-las já que o usuário irá vê-las assim que entrar na WebApp. Serão fornecidos links gráficos? Onde será exibido o menu de navegação? Que outras informações o usuário verá? Perguntas como essas devem ser respondidas pelo protótipo.

7.5.6 Modelo funcional para WebApps

Muitas WebApps oferecem uma ampla gama de funções computacionais e manipuladoras que podem ser associadas diretamente ao conteúdo (seja usando-o, seja produzindo-o) e que

¹⁶ Diagramas de sequência e diagramas de estados são modelados usando-se a notação da UML. Os diagramas de estados são descritos na Seção 7.3. Veja o Apêndice 1 para mais detalhes.

frequentemente são sua meta principal de interação com o usuário. Por essa razão, os requisitos funcionais têm de ser analisados e, quando necessário, modelados.

O *modelo funcional* lida com dois elementos de processamento da WebApp, cada um dos quais representando um diferente nível de abstração procedural: (1) funcionalidade observável pelo usuário fornecida pela WebApp aos usuários finais e (2) as operações contidas nas classes de análise que implementam comportamentos associados à classe.

A funcionalidade observável pelos usuários engloba quaisquer funções de processamento iniciadas diretamente pelo usuário. Uma WebApp financeira poderia implementar uma série de funções financeiras (por exemplo, uma calculadora de crédito educativo para o ensino superior ou uma calculadora para planos de aposentadoria). Essas funções poderiam, na verdade, ser implementadas usando-se operações dentro das classes de análise, porém, do ponto de vista do usuário final, a função (mais precisamente, os dados fornecidos pela função) é o resultado visível.

Em um nível de abstração procedural mais baixo, o modelo de requisitos descreve o processamento a ser realizado pelas operações das classes de análise. Essas operações manipulam atributos de classes e estão envolvidas, já que as classes colaboram entre si para cumprir determinado comportamento exigido.

Independentemente do nível de abstração procedural, o diagrama de atividades UML pode ser utilizado para representar detalhes do processamento. No nível de análise, os diagramas de atividades devem ser usados apenas onde a funcionalidade é relativamente complexa. Grande parte da complexidade de muitas WebApps ocorre não na funcionalidade fornecida, mas sim na natureza das informações que podem ser acessadas e nas maneiras pelas quais podem ser manipuladas.

Um exemplo de funcionalidade relativamente complexa do **CasaSeguraGarantida.com** é tratado por um caso de uso intitulado *Obter recomendações para a disposição dos sensores no espaço disponível*. O usuário já desenvolveu um layout para o espaço a ser monitorado e, nesse caso de uso, escolhe esse layout e solicita posições recomendadas para os sensores de acordo com o layout. **CasaSeguraGarantida.com** responde com uma representação gráfica do layout com informações adicionais sobre os pontos recomendados para posicionamento dos sensores. A interação é bastante simples, o conteúdo é ligeiramente mais complexo, porém a funcionalidade subjacente é muito sofisticada. O sistema deve empreender uma análise relativamente complexa do layout do andar para determinar o conjunto ótimo de sensores. Ele tem de examinar as dimensões dos ambientes, a posição das portas e janelas e coordená-las com as capacidades e especificações dos sensores. Nada trivial! Um conjunto de diagramas de atividades pode ser usado para descrever o processamento para esse caso de uso.

O segundo exemplo é o caso de uso *Controlar câmeras*. Neste, a interação é relativamente simples, porém há grande risco de funcionalidade complexa, dado que essa “simples” operação requer comunicação complexa com os dispositivos localizados remotamente e acessíveis via Internet. Uma outra possível complicação está relacionada com a negociação do controle, quando várias pessoas autorizadas tentam monitorar e/ou controlar um único sensor ao mesmo tempo.

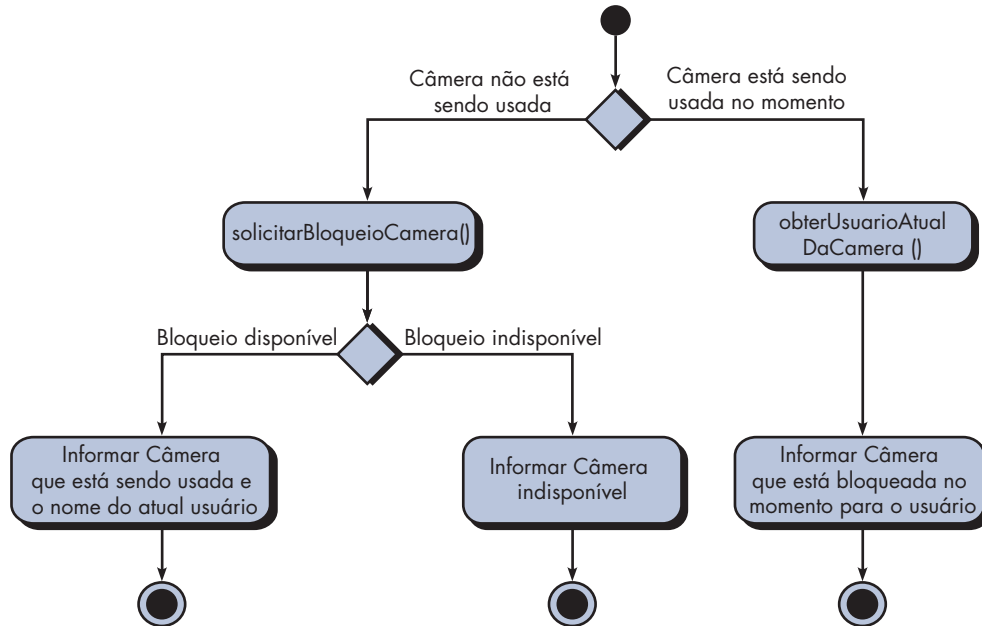
A Figura 7.11 ilustra um diagrama de atividades para a operação *assumirControleDaCamera()* que faz parte da classe de análise **Camera** usada no caso de uso *Controlar cameras*. Deve-se notar que duas operações adicionais são chamadas dentro do fluxo procedural: *solicitarBloqueioCamera()*, que tenta bloquear a câmera para esse usuário e *obterUsuarioCameraAtual()*, que recupera o nome do usuário que está controlando a câmera no momento. Os detalhes construtivos indicando como essas operações são chamadas, bem como os detalhes da interface para cada operação, não são considerados até que o projeto da WebApp seja iniciado.

7.5.7 Modelos de configuração para WebApps

Em alguns casos, o modelo de configuração nada mais é que uma lista de atributos no servidor e no cliente. Entretanto, para WebApps mais complexas, uma série de detalhes de configuração

FIGURA 7.11

Diagrama de atividades para a operação `assumirControleDaCamera()`



(por exemplo, distribuição da carga entre vários servidores, arquiteturas de caching, bancos de dados remotos, vários servidores atendendo vários objetos na mesma página Web) poderiam ter um impacto na análise e no projeto. O *diagrama de disponibilização da UML* pode ser utilizado em situações em que complexas arquiteturas de configuração têm de ser consideradas.

Para **CasaSeguraGarantida.com**, a funcionalidade e conteúdo público deveriam ser especificados como acessíveis a todos os principais clientes Web (isto é, aqueles com mais do que 1% de participação no mercado¹⁷). Ao contrário, pode ser aceitável restringir a funcionalidade de monitoramento e controle mais complexa (que poderia ser acessado somente pelos usuários **Proprietário**) para um conjunto de clientes menor. O modelo de configuração para **CasaSeguraGarantida.com** também especificará a interoperabilidade com aplicações de monitoramento e bancos de dados de produtos existentes.

7.5.8 Modelo de navegação

O modelo de navegação considera a maneira como cada categoria de usuário irá navegar de um elemento da WebApp (por exemplo, objeto de conteúdo) para outro. A mecânica de navegação é definida como parte do projeto. Nesse estágio, devemos nos concentrar nos requisitos gerais de navegação. As seguintes perguntas devem ser consideradas:

- Deveriam certos elementos ser mais fáceis de ser acessados (exigir um número de passos de navegação menor) do que outros? Qual a prioridade para a apresentação?
- Deveriam certos elementos ser enfatizados para forçar os usuários a navegar em sua direção?
- Como os erros de navegação deveriam ser tratados?
- Deveria a navegação para grupos de elementos relacionados ter maior prioridade em relação à navegação para um elemento específico?
- A navegação deveria ser obtida via links, via acesso baseado em pesquisa ou por outros meios?

¹⁷ Determinar a participação de mercado para navegadores é notoriamente problemático e varia, dependendo de qual tipo de pesquisa for utilizada. Não obstante, na época em que este livro foi escrito, o Internet Explorer e o Firefox eram os únicos navegadores que ultrapassavam os 30%, enquanto o Mozilla, o Opera e o Safari eram os únicos consistentemente acima de 1%.

- Deveriam certos elementos ser apresentados aos usuários no contexto de ações de navegação prévias?
- Deveria o log de navegação ser mantido para os usuários?
- Deveria existir um menu ou mapa de navegação completo (em vez de um simples link “de retorno” ou ponteiro indicando direção) em qualquer ponto da interação de um usuário?
- Deveria o projeto de navegação ser dirigido pelos comportamentos de usuário mais comumente esperados ou pela importância percebida dos elementos definidos da WebApp?
- Poderia um usuário “armazenar” sua navegação prévia pela WebApp para agilizar seu uso futuro?
- Para qual categoria de usuário a navegação otimizada deveria ser desenvolvida?
- Como os links externos à WebApp deveriam ser tratados? Sobrepondo a janela do navegador existente? Como uma nova janela do navegador? Como um quadro separado?

Essas e muitas outras perguntas devem ser feitas e respondidas como parte da análise de navegação.

Você e outros interessados também devem determinar os requisitos gerais para navegação. Por exemplo, será fornecido um “mapa do site” para dar aos usuários uma visão geral de toda a estrutura da WebApp? Será possível um usuário fazer um “tour guiado” que destacará os elementos mais importantes (objetos de conteúdo e funções) disponíveis? Um usuário será capaz de acessar objetos de conteúdo ou funções baseadas em atributos definidos daqueles elementos (por exemplo, um usuário poderia querer ter acesso a todas as fotografias de determinado prédio ou a todas as funções que possibilitam o cálculo de peso)?

7.6 RESUMO

Os modelos orientados a fluxos se concentram no fluxo de objetos de dados à medida que são transformados por funções de processamento. Extraídos da análise estruturada, os modelos orientados a fluxos usam o diagrama de fluxo de dados, uma notação de modelagem que representa como a entrada é transformada em saída à medida que os objetos de dados se deslocam através de um sistema. Cada função de software que transforma dados é descrita por uma narrativa ou especificação de processos. Além do fluxo de dados, esse elemento de modelagem também representa o fluxo de controle — uma representação que ilustra como os eventos afetam o comportamento de um sistema.

A modelagem comportamental representa o comportamento dinâmico. O modelo comportamental usa entrada de elementos baseados em cenários, orientados a fluxos e baseados em classes para representar os estados das classes de análise e do sistema como um todo. Para tanto, são identificados os estados, são definidos os eventos que fazem com que uma classe (ou o sistema) passe por uma transição de um estado para outro e também são identificadas as ações que ocorrem à medida que acontece a transição. Os diagramas de estados e os diagramas de sequência são a notação utilizada para modelagem comportamental.

Os padrões de análise possibilitam a um engenheiro de software usar conhecimento do domínio existente para facilitar a criação de um modelo de requisitos. Um padrão de análise descreve uma função ou recurso de software específico que pode ser descrito por meio de um conjunto de casos de uso coerente. Ele especifica o intuito do padrão, o motivo para seu emprego, restrições que limitam seu uso, sua aplicabilidade em vários domínios de problemas, a estrutura geral do padrão, seu comportamento e colaborações, bem como outras informações complementares.

A modelagem de requisitos para WebApps pode usar a maioria, se não todos, dos elementos de modelagem discutidos neste livro. Entretanto, tais elementos são aplicados em um conjunto de modelos especializados que tratam o conteúdo, interação, função, navegação e a configuração cliente/servidor em que a WebApp reside.

PROBLEMAS E PONTOS A PONDERAR

- 7.1. Qual a diferença fundamental entre as estratégias de análise estruturada e orientadas a objetos para a análise de requisitos?
- 7.2. Em um diagrama de fluxo de dados, uma seta representa um fluxo de controle ou algo mais?
- 7.3. O que é “continuidade de fluxo de informações” e como se aplica à medida que um diagrama de fluxo de dados é refinado?
- 7.4. Como a análise sintática é usada na criação de um DFD?
- 7.5. O que é uma especificação de controle?
- 7.6. Uma PSPEC e um caso de uso são a mesma coisa? Se não forem, explique as diferenças.
- 7.7. Existem dois tipos de “estados” que os modelos comportamentais são capazes de representar. Quais são eles?
- 7.8. Como um diagrama de sequência difere de um diagrama de estado? Em que são similares?
- 7.9. Sugira três padrões de requisitos para um celular moderno e redija uma breve descrição de cada um deles. Poderiam esses padrões ser utilizados para outros dispositivos? Cite um exemplo.
- 7.10. Escolha um dos padrões que você desenvolveu no Problema 7.9 e faça uma descrição de padrão relativamente completa similar em conteúdo e estilo àquela apresentada na Seção 7.4.2.
- 7.11. Que nível de modelagem de análise você acredita que seria necessária para **Casa-SeguraGarantida.com**? Seria necessário cada um dos tipos de modelos descritos na Seção 7.5.3?
- 7.12. Qual o objetivo do modelo de interações para uma WebApp?
- 7.13. Poder-se-ia argumentar que um modelo funcional para WebApp deveria ser postergado até a fase de projeto. Apresente os prós e os contras desse argumento.
- 7.14. Qual o objetivo de um modelo de configuração?
- 7.15. Em que o modelo de navegação difere do modelo de interações?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Foram publicados dezenas de livros sobre análise estruturada. Todos cobrem o tema de forma adequada, porém poucos o fazem de maneira realmente excelente. DeMarco e Plauger (*Structured Analysis and System Specification*, Pearson, 1985) é um clássico que ainda se mantém como uma boa introdução à notação básica. Livros como os de Kendall e Kendall (*Systems Analysis and Design*, 5. ed., Prentice-Hall, 2002), Hoffer et al. (*Modern Systems Analysis and Design*, Addison-Wesley, 3. ed., 2001), Davis e Yen (*The Information System Consultant's Handbook: Systems Analysis and Design*, CRC Press, 1998) e Modell (*A Professional's Guide to Systems Analysis*, 2. ed., McGraw-Hill, 1996) são referências proveitosas. O livro de Yourdon (*Modern Structured Analysis*, Yourdon-Press, 1989) sobre o tema ainda se encontra entre os mais completos publicados até hoje.

A modelagem comportamental apresenta uma visão dinâmica importante do comportamento de um sistema. Livros como os de Wagner e seus colegas (*Modeling Software with Finite State Machines: A Practical Approach*, Auerbach, 2006) e Boerger e Staerk (*Abstract State Machines*, Springer, 2003) apresentam uma discussão abrangente sobre diagramas de estado e outras representações comportamentais.

A maioria dos livros escritos que abordam o tema padrões de software enfoca o projeto de software. Entretanto, livros como os de Evans (*Domain-Driven Design*, Addison-Wesley, 2003) e Fowler ([Fow03] e [Fow97]) examinam especificamente os padrões de análise.

Um tratado aprofundado sobre a modelagem de análise para WebApps é apresentado por Pressman e Lowe [Pre08]. Artigos contidos em uma antologia editada por Murugesan e Desphande (*Web Engineering: Managing Diversity and Complexity of Web Application Development*, Springer, 2001) discutem vários aspectos dos requisitos de uma WebApp. Além destes, o *Proceedings of the International Conference on Web Engineering* publicado anualmente trata regularmente de questões referentes à modelagem de requisitos.

Uma ampla gama de fontes de informação sobre modelagem de requisitos se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevante para a modelagem de análise, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CONCEITOS DE PROJETO

CONCEITOS-
-CHAVE

abstração	212
arquitetura	213
aspectos	217
atributos de qualidade	210
bom projeto	210
coesão	216
diretrizes de qualidade	209
encapsulamento de informações	215
independência funcional	217
modularidade	214
padrões	214
processo de projeto	209
projeto de dados	209
projeto de software	211
projeto orientado a objetos	218
refatoração	218
refinamento gradual	217
separação de interesses	214

A atividade de projeto de software¹ engloba o conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto com alta qualidade. Os princípios de projeto estabelecem uma filosofia que prevalece sobre as atitudes e ações do desenvolvimento, orientando as atividades para realizar o projeto. Os conceitos de projeto devem ser estabelecidos e entendidos antes de aplicar a prática de projeto, que deve levar à criação de várias representações do software que servem como um guia para a atividade de construção que se segue.

A atividade projeto é crítica para o êxito em engenharia de software. No início dos anos 1990, Mitch Kapor, o criador do Lotus 1-2-3, apresentou um “manifesto de projeto de software” no *Dr. Dobbs Journal*. Disse ele:

O que é projeto? É onde você fica com o pé em dois mundos — o mundo da tecnologia e o mundo das pessoas e dos propósitos do ser humano — e você tenta unir os dois...

O crítico de arquitetura romano, Vitruvius, lançou a noção de que prédios bem projetados eram aqueles que apresentavam solidez, comodidade e deleite. O mesmo poderia ser dito em relação a software de boa qualidade. *Solidez*: um programa não deve apresentar nenhum bug que impeça seu funcionamento. *Comodidade*: um programa deve ser adequado aos propósitos para os quais foi planejado. *Deleite*: a experiência de usar o programa deve ser prazerosa. Temos aqui os princípios de uma teoria de projeto de software.

PANORAMA

O que é? Projeto é o que quase todo engenheiro quer fazer. É o lugar onde a criatividade impera — onde os requisitos dos interessados, as necessidades da aplicação e considerações técnicas se juntam na formulação de um produto ou sistema. O projeto cria uma representação ou modelo do software, mas diferentemente do modelo de requisitos (que se concentra na descrição do “O que” é para ser feito: dos dados, função e comportamento necessários), o modelo de projeto indica “O Como” fazer, fornecendo detalhes sobre a arquitetura de software, estruturas de dados, interfaces e componentes fundamentais para implementar o sistema.

Quem realiza? Os engenheiros de software conduzem cada uma das tarefas de projeto.

Por que é importante? O projeto permite que se modele o sistema ou produto a ser construído. O modelo pode ser avaliado em termos de qualidade e aperfeiçoado ANTES de o código ser gerado, ou de os testes serem realizados, ou de os usuários finais se envolverem com grandes números. Projeto é o lugar onde a qualidade do software é estabelecida.

Quais são as etapas envolvidas? O projeto representa o software de várias formas diferentes.

Primeiramente, a arquitetura do sistema ou do produto tem de ser representada. Em seguida, são modeladas as interfaces que conectam o software aos usuários finais a outros sistemas e a dispositivos, bem como seus próprios componentes internos. Por fim, os componentes de software usados para construir o sistema são projetados. Cada uma dessas visões representa uma diferente ação de projeto, mas todas devem estar de acordo com um conjunto de conceitos básicos de projeto que orientam o trabalho de projeto de software.

Qual é o artefato? Um modelo de projeto que engloba representações de arquitetura, interface, no nível de componentes e de utilização é o principal artefato gerado durante o projeto de software.

Como garantir que o trabalho foi realizado corretamente? O modelo de projeto é avaliado pela equipe de software em um esforço para determinar se ele contém erros, inconsistências ou omissões; se existem alternativas melhores; e se o modelo pode ser implementado de acordo com as restrições, prazo e orçamento estabelecidos.

1 N. de R.T.: o termo em inglês dessa atividade é *design*. Refere-se à atividade de engenharia cujo foco é definir “como” os requisitos estabelecidos do projeto devem ser implementados no software. É uma fase que se apresenta de maneira similar nas diversas especializações da engenharia como a Civil, Naval, Química e Mecânica.

O objetivo da atividade projetar é gerar um modelo ou representação que apresente solidez, comodidade e deleite. Para tanto, temos de praticar a diversificação e, depois, a convergência. Belady [Bel81] afirma que “diversificação é a aquisição de um repertório de alternativas, a matéria-prima do projeto: componentes, soluções de componentes e conhecimento, todos contidos em catálogos, livros-textos e na mente”. Assim que esse conjunto diversificado de informações for montado, temos de escolher elementos do repertório que atendam os requisitos definidos pela engenharia de requisitos e pelo modelo de análise (Capítulos 5 ao 7). À medida que isso ocorre, são consideradas e rejeitadas alternativas e convergimos para “uma particular configuração de componentes e, portanto, a criação do produto final” [Bel81].

Diversificação e convergência combinam intuição e julgamento baseado na experiência de construção de entidades similares, um conjunto de princípios e/ou heurística que orientam a maneira por meio da qual o modelo evolui, um conjunto de critérios que permitem que a qualidade seja avaliada e um processo de iteração que, ao fim, leva a uma representação final do projeto. O projeto de software muda continuamente à medida que novos métodos, melhor análise e entendimento mais abrangente evoluem.² Mesmo hoje em dia, a maioria das metodologias de projeto de software carece da profundidade, flexibilidade e natureza quantitativa que normalmente estão associadas às disciplinas mais clássicas de engenharia de projeto. Entretanto, existem efetivamente métodos para projeto de software, critérios para qualidade de projeto estão disponíveis e notação de projeto pode ser aplicada. Neste capítulo, exploraremos os conceitos e princípios fundamentais aplicáveis a todos os projetos de software, os elementos do modelo de projeto e o impacto dos padrões no processo de projeto. Nos Capítulos 9 a 13 apresentaremos uma série de métodos de projeto de software à medida que são aplicados ao projeto da arquitetura, de interfaces e de componentes, bem como as abordagens de projeto baseada em padrões e orientada para a Web.

8.1 PROJETO NO CONTEXTO DA ENGENHARIA DE SOFTWARE

“O milagre mais comum da engenharia de software é a transição da análise para o projeto e do projeto para o código.”

Richard Due’



O projeto de software sempre deve começar levando em consideração os dados — a base para todos os demais elementos do projeto. Após estabelecida a base, a arquitetura tem de ser extraída. Só então devem se realizar outras tarefas de projeto.

O projeto de software reside no núcleo técnico da engenharia de software e é aplicado independentemente do modelo de processos de software utilizado. Iniciando assim que os requisitos de software tiverem sido analisados e modelados, o projeto de software é a última ação da engenharia de software da atividade de modelagem e prepara a cena para a **construção** (geração de código e testes).

Cada um dos elementos do modelo de requisitos (Capítulos 6 e 7) fornece informações necessárias para criar os quatro modelos de projeto necessários para uma especificação completa. O fluxo de informações durante o projeto de software é ilustrado na Figura 8.1. O modelo de requisitos, manifestado por elementos baseados em cenários, baseados em classes, orientado a fluxos e comportamentais, alimenta a tarefa de projeto. Usando a notação de projeto e os métodos de projeto discutidos em capítulos posteriores, o projeto gera um projeto de dados/classes, um projeto de arquitetura, um projeto de interfaces e um projeto de componentes.

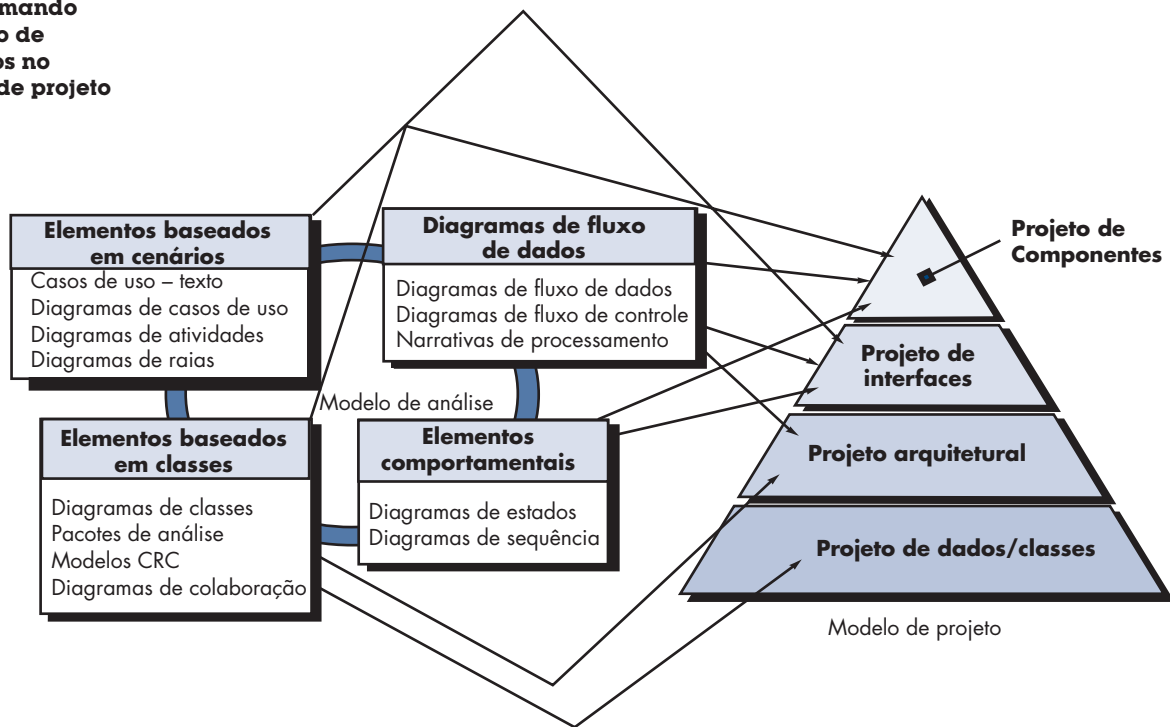
O projeto de dados/classes transforma os modelos de classes (Capítulo 6) em realizações de classes de projeto e nas estruturas de dados dos requisitos necessárias para implementar o software. Os objetos e os relacionamentos definidos nos cartões CRC e no conteúdo detalhado dos dados representados por atributos de classes e outra notação fornecem a base para a realização do projeto de dados. Parte do projeto de classes pode ocorrer com o projeto da arquitetura de software. O projeto de classe mais detalhado ocorre à medida que cada componente de software é projetado.

O projeto arquitetural define os relacionamentos entre os principais elementos estruturais do software, os estilos arquiteturais e padrões de projeto que podem ser usados para satisfazer os requisitos definidos para o sistema e as restrições que afetam o modo pelo qual a arquitetura

² Aqueles leitores com maior interesse na filosofia do projeto de software talvez se interessem pela intrigante discussão de Philippe Kruchten sobre o projeto “pós-moderno” [Kru05a].

FIGURA 8.1

**Transformando
o modelo de
requisitos no
modelo de projeto**



pode ser implementada [Sha96]. A representação do projeto arquitetural — a organização da solução técnica de um sistema baseado em computador — é derivada do modelo de requisitos.

O projeto de interfaces descreve como o software se comunica com sistemas que interrompem com ele e com as pessoas que o utilizam. Uma interface implica fluxo de informações (por exemplo, dados e/ou controle) e um tipo de comportamento específico. Consequentemente, modelos comportamentais e de cenários de uso fornecem grande parte das informações necessárias para o projeto de interfaces.

O projeto de componentes transforma elementos estruturais da arquitetura de software em uma descrição procedural dos componentes de software. As informações obtidas dos modelos baseados em classes, modelos de fluxo e modelos comportamentais servem como base para o projeto de componentes.

Durante o projeto tomamos decisões que, em última análise, afetarão o sucesso da construção do software e, igualmente importante, a facilidade de manutenção do software. Mas por que o projeto é tão importante?

A importância do projeto de software pode ser definida em uma única palavra — *qualidade*. Projeto é a etapa em que a qualidade é incorporada na engenharia de software. O projeto nos fornece representações de software que podem ser avaliadas em termos de qualidade. Projeto é a única maneira em que podemos traduzir precisamente os requisitos dos interessados em um produto ou sistema de software finalizado. O projeto de software serve como base para todas as atividades de apoio e da engenharia de software que seguem. Sem um projeto, corremos o risco de construir um sistema instável — um sistema que falhará quando forem feitas pequenas alterações; um sistema que talvez seja difícil de ser testado; um sistema cuja qualidade não pode ser avaliada até uma fase avançada do processo de software, quando o tempo está se esgotando e muito dinheiro já foi gasto.

“Há duas maneiras de construir um projeto de software. Uma delas é fazê-lo tão simples que, obviamente, não existirão deficiências, e a outra maneira é fazê-lo tão complicado que não há nenhuma deficiência óbvia. O primeiro método é bem mais difícil.”

C. A. R. Hoare

CASA SEGURA

*Projeto versus codificação*

Cena: Sala do Ed, enquanto a equipe se prepara para traduzir os requisitos para o projeto.

Atores: Jamie, Vinod e Ed — todos membros da equipe de engenharia de software do CasaSegura.

Conversa:

Jamie: Você sabe, Doug [o gerente da equipe] está obcecado pelo projeto. Tenho que ser honesto; o que eu realmente adoro fazer é programar. Dê-me o C++ ou Java e ficarei feliz.

Ed: Nada... Você gosta de projetar.

Jamie: Você não está me ouvindo; programar é o canal.

Vinod: Acredito que aquilo que o Ed quis dizer é que você realmente não gosta de programar; você gosta de projetar e expressar isto na forma de código de programa. Código é a linguagem que você usa para representar um projeto.

Jamie: E o que há de errado nisso?

Vinod: Nível de abstração.

Jamie: Hã?

Ed: Uma linguagem de programação é boa para representar detalhes como estruturas de dados e algoritmos, mas não é tão boa para representar a colaboração componente-componente ou a arquitetura... Coisas do tipo.

Vinod: E uma arquitetura ferrada pode arruinar até mesmo o melhor código.

Jamie (pensando por um minuto): Portanto, você está dizendo que não posso representar arquitetura no código... Isso não é verdade.

Vinod: Certamente você pode envolver arquitetura no código, mas na maioria das linguagens de programação, é bem difícil ter uma visão geral rápida da arquitetura examinando-se o código.

Ed: E é isso que queremos antes de começar a programar.

Jamie: OK, talvez projetar e programar sejam coisas diferentes, mas ainda assim prefiro programar.

8.2 O PROCESSO DE PROJETO

O projeto de software é um processo iterativo através do qual os requisitos são traduzidos em uma “planta” para construir o software. Inicialmente, a planta representa uma visão holística do software. O projeto é representado em um alto nível de abstração — um nível que pode ser associado diretamente ao objetivo específico do sistema e aos requisitos mais detalhados de dados, funcionalidade e comportamento. À medida que ocorrem as iterações do projeto, refinamento subsequente leva a representações do projeto em níveis de abstração cada vez mais baixos. Estes ainda podem ser associados aos requisitos, mas a conexão é mais sutil.

8.2.1 Diretrizes e atributos da qualidade de software

Ao longo do processo de projeto, a qualidade do projeto que evolui é avaliada com uma série de revisões técnicas discutidas no Capítulo 15. McGlaughlin [McG91] sugere três características que servem como um guia para a avaliação de um bom projeto:

- O projeto deve implementar todos os requisitos explícitos contidos no modelo de requisitos e deve acomodar todos os requisitos implícitos desejados pelos interessados.
- O projeto deve ser um guia legível, compreensível para aqueles que geram código e para aqueles que testam e, subsequentemente, dão suporte ao software.
- O projeto deve dar uma visão completa do software, tratando os domínios de dados, funcional e comportamental sob a perspectiva de implementação.

Cada uma dessas características é, na verdade, uma meta do processo de projeto. Mas como cada uma é alcançada?

Diretrizes de qualidade. Para avaliar a qualidade da representação de um projeto, você e outros membros da equipe de software devem estabelecer critérios técnicos para um bom projeto. Na Seção 8.3, discutimos conceitos de projeto que também servem como critérios de qualidade de software. Por enquanto, consideremos as seguintes diretrizes:

“Escrever um trecho de código inteligente que funcione é uma coisa; projetar algo que possa dar suporte a negócios duradouros é outra totalmente diferente.”

C. Ferguson

? Quais são as características de um bom projeto?

1. Um projeto deve exibir uma arquitetura que (1) foi criada usando estilos ou padrões arquiteturais reconhecíveis, (2) seja composta por componentes que apresentem boas características de projeto (discutidos mais adiante neste capítulo) e (3) possa ser implementada de uma forma evolucionária³ facilitando, portanto, a implementação e os testes.
2. Um projeto deve ser modular; o software deve ser logicamente particionado em elementos ou subsistemas, de modo que seja fácil de testar e manter.
3. Um projeto deve conter representações distintas de: dados, arquitetura, interfaces e componentes.
4. Um projeto deve levar as estruturas de dados adequadas às classes a ser implementadas e baseadas em padrões de dados reconhecíveis.
5. Um projeto deve levar a componentes que apresentem características funcionais independentes (baixo acoplamento).
6. Um projeto deve levar a interfaces que reduzam a complexidade das conexões entre os componentes e o ambiente externo (encapsulamento).
7. Um projeto deve ser obtido usando-se um método repetível, isto é, dirigido por informações obtidas durante a análise de requisitos de software.
8. Um projeto deve ser representado usando-se uma notação que efetivamente comunique seu significado.

Essas diretrizes não são atingidas por acaso. Elas são alcançáveis por meio da aplicação de princípios de projeto fundamentais, de metodologia sistemática e de revisão.

Atributos de qualidade. A Hewlett-Packard [Gra87] desenvolveu um conjunto de atributos de qualidade de software ao qual foi atribuído o acrônimo FURPS — *functionality* (funcionalidade), *usability* (usabilidade), *reliability* (confiabilidade), *performance* (desempenho) e *supportability* (facilidade de suporte). Os atributos de qualidade FURPS representam uma meta para todo projeto de software:

- A *funcionalidade* é avaliada pela observação do conjunto de características e capacidades do programa, a generalidade das funções que são entregues e a segurança do sistema como um todo.

“Qualidade não é algo que se coloque sobre os assuntos e objetos como um enfeite em uma árvore de Natal.”

Robert Pirsig

INFORMAÇÕES



Avaliação da qualidade do projeto — a revisão técnica

O projeto é importante porque permite à equipe de software avaliar a qualidade⁴ do software antes de ser implementado — em um momento em que os erros, as omissões ou as inconsistências são fáceis e baratas de ser corrigidos. Mas como avaliar a qualidade durante um projeto? O software não pode ser testado, pois não existe nenhum software executável para testar. O que fazer então?

Durante um projeto, a qualidade é avaliada realizando-se uma série de revisões técnicas (technical reviews, TRs). As TRs são discutidas em detalhes no Capítulo 15⁵ mas vale fazer um resumo neste momento. A revisão técnica é uma reunião conduzida por membros da equipe de software. Normalmente duas, três ou quatro pessoas participam, dependendo do escopo das informações de projeto a ser revisadas. Cada pessoa desempe-

nha um papel: um *líder de revisão* planeja a reunião, estabelece uma agenda e conduz a reunião; o *registrador* toma notas de modo que nada seja perdido; o *produtor* é a pessoa cujo artefato (por exemplo, o projeto de um componente de software) está sendo revisado. Antes de uma reunião, cada pessoa da equipe de revisão recebe uma cópia do artefato do projeto e lhes é solicitada sua leitura, procurando encontrar erros, omissões ou ambiguidade. Quando a reunião começa, o intuito é perceber todos os problemas com o artefato, de modo que possam ser corrigidos antes da implementação começar. A TR dura, tipicamente, entre 90 minutos e 2 horas. Na conclusão, a equipe de revisão determina se outras ações são necessárias por parte do produtor antes de o artefato do projeto ser aprovado como parte do modelo de projeto final.

3 Evolucionária: incremental, com entregas por partes. Para sistemas menores, algumas vezes o projeto pode ser desenvolvido linearmente.

4 Os fatores de qualidade discutidos no Capítulo 23 podem ajudar a equipe de revisão à medida que avalia a qualidade.

5 Você deve considerar consulta do Capítulo 15 neste momento. As revisões técnicas são parte crítica do processo de projeto e é um importante mecanismo para atingir qualidade em um projeto.



Os projetistas de software tendem a se concentrar no problema a ser resolvido. Apenas não se esqueça de que os atributos de qualidade FURPS sempre fazem parte do problema. Eles têm de ser considerados.

- A *usabilidade* é avaliada considerando-se fatores humanos (Capítulo 11), estética, consistência e documentação como um todo.
- A *confiabilidade* é avaliada medindo-se a frequência e a severidade das falhas, a precisão dos resultados gerados, o tempo médio entre defeitos (*mean-time-to-failure*, MTTF), a capacidade de se recuperar de uma falha e a previsibilidade do programa.
- O *desempenho* é medido considerando-se a velocidade de processamento, o tempo de resposta, o consumo de recursos, vazão (*throughput*) e eficiência.
- A *facilidade de suporte* combina a habilidade de estender o programa (extensibilidade), a adaptabilidade e a reparabilidade — esses três atributos representam um termo mais comum, *facilidade de manutenção* — e, além disso, a facilidade de realizar testes, a compatibilidade, a facilidade de configurar (a habilidade de organizar e controlar elementos da configuração do software, Capítulo 22), a facilidade com a qual um sistema pode ser instalado, bem como a facilidade com a qual os problemas podem ser localizados.

Nem todo atributo de qualidade de software tem o mesmo peso à medida que o projeto é desenvolvido. Uma aplicação poderia enfatizar a funcionalidade com ênfase especial na segurança. Outra poderia demandar desempenho com particular ênfase na velocidade de processamento. Um terceiro foco poderia ser na confiabilidade. Independentemente do peso dado, é importante notar que esses atributos de qualidade devem ser considerados quando o projeto se inicia, e não após o projeto estar completo e a construção tiver começado.

8.2.2 A evolução de um projeto de software

A evolução de um projeto de software é um processo contínuo que já atinge quase seis décadas. Os trabalhos iniciais concentravam-se em critérios para o desenvolvimento de programas modulares [Den73] e de métodos para refinamento das estruturas de software de uma forma *topdown* [Wir71]. Aspectos procedurais da definição de um projeto evoluíram e convergiram para uma filosofia denominada *programação estruturada* [Dah72], [Mil72]. Trabalhos posteriores propuseram métodos para a tradução de fluxos de dados [Ste74] ou da estrutura de dados (por exemplo, [Jac75], [War74]) em uma definição de projeto. Abordagens de projeto mais recentes (por exemplo, [Jac92], [Gam95]) propuseram uma abordagem orientada a objetos para a derivação do projeto. Atualmente, a ênfase em projeto de software tem sido na arquitetura de software [Kru06] e nos padrões de projeto que podem ser utilizados para implementar arquiteturas de software e níveis de abstração de projeto mais baixos (por exemplo, [Hol06] [Sha05]). Tem crescido a ênfase em métodos orientados a aspectos (por exemplo, [Cla05], [Jac04]), no desenvolvimento dirigido a modelos [Sch06] e dirigido a testes [Ast04] que enfatizam técnicas para se atingir uma modularidade e estrutura arquitetural mais efetiva nos projetos criados.

Uma série de métodos de projetos, originários dos trabalhos citados, está sendo aplicada em toda a indústria. Assim como os métodos de análise apresentados nos Capítulos 6 e 7, cada método de projeto de software introduz heurística e notação únicas, bem como uma visão um tanto provinciana daquilo que caracteriza a qualidade de um projeto. Mesmo assim, todos os métodos possuem uma série de características comuns: (1) um mecanismo para a tradução do modelo de requisitos em uma representação de projeto, (2) uma notação para representar componentes funcionais e suas interfaces, (3) heurística para refinamento e particionamento e (4) diretrizes para avaliação da qualidade.

Independentemente do método de projeto utilizado, devemos aplicar um conjunto de conceitos básicos ao projeto de dados, de arquitetura, de interface e dos componentes. Tais conceitos são considerados nas seções a seguir.

“Um projetista sabe que seu projeto atingiu a perfeição não quando não resta mais nada a ser acrescentado, mas quando não há mais nada a ser eliminado.”

Antoine de St-Exupéry



Quais características são comuns a todos os métodos de projeto?

CONJUNTO DE TAREFAS

**Conjunto de tarefas genéricas para projeto**

1. Examinar o modelo do domínio de informação e projetar estruturas de dados apropriadas para objetos de dados e seus atributos.
2. Usar o modelo de análise, selecionar um estilo de arquitetura apropriado ao software.
3. Dividir o modelo de análise em subsistemas de projeto e alocá-los na arquitetura:
 - Certificar-se de que cada subsistema seja funcionalmente coeso.
 - Projetar interfaces de subsistemas.
 - Alocar classes ou funções de análise para cada subsistema.
4. Criar um conjunto de classes ou componentes de projeto:
 - Traduzir a descrição de classes de análise em uma classe de projeto.
 - Verificar cada classe de projeto em relação aos critérios de projeto; considerar questões de herança.
 - Definir métodos e mensagens associadas a cada classe de projeto.
 - Avaliar e selecionar padrões de projeto para uma classe ou um subsistema de projeto.
5. Rever as classes de projeto e revisar quando necessário.
5. Projetar qualquer interface necessária para sistemas ou dispositivos externos.
6. Projetar a interface do usuário:
 - Revisar os resultados da análise de tarefas.
 - Especificar a sequência de ações baseando-se nos cenários de usuário.
 - Criar um modelo comportamental da interface.
 - Definir objetos de interface, mecanismos de controle.
 - Rever o projeto de interfaces e revisar quando necessário.
7. Conduzir o projeto de componentes.
 - Especificar todos os algoritmos em um nível de abstração relativamente baixo.
 - Refinar a interface de cada componente.
 - Definir estruturas de dados dos componentes.
 - Revisar cada componente e corrigir todos os erros descobertos.
8. Desenvolver um modelo de implantação.

8.3 CONCEITOS DE PROJETO

Um conjunto de conceitos fundamentais de projeto de software evoluiu ao longo da história da engenharia de software. Embora o grau de interesse em cada conceito tenha variado ao longo dos anos, cada um resistiu ao tempo. Esses conceitos fornecem ao projetista de software uma base a partir da qual métodos de projeto mais sofisticados podem ser aplicados. Ajudam-nos a responder as seguintes questões:

- Quais critérios podem ser usados para particionar o software em componentes individuais?
- Como os detalhes de função ou estrutura de dados são separados de uma representação conceitual do software?
- Quais critérios uniformes definem a qualidade técnica de um projeto de software?

M. A. Jackson [Jac75] disse uma vez: “O princípio da sabedoria para um [engenheiro de software] é reconhecer a diferença entre fazer um programa funcionar e fazer com que ele funcione corretamente”. Os conceitos fundamentais de projeto de software fornecem a organização necessária para estruturá-la e para “fazer com que ele funcione corretamente”.

Nas seções a seguir, apresentamos uma breve visão de importantes conceitos de projeto de software que englobam tanto o desenvolvimento de software tradicional quanto o orientado a objetos.

8.3.1 Abstração

Ao se considerar uma solução modular para qualquer problema, muitos níveis de abstração podem se apresentar. No nível de abstração mais alto, uma solução é expressa em termos abrangentes usando a linguagem do domínio do problema. Em níveis de abstração mais baixos, uma descrição mais detalhada da solução é fornecida. A terminologia do domínio do problema é associada à terminologia de implementação para definir uma solução. Por fim, no nível de

“Abstração é uma das maneiras fundamentais como nós, seres humanos, lidamos com a complexidade.”

Grady Booch



Como projetista, trabalhe arduamente para derivar tanto as abstrações procedurais quanto a de dados que atendam ao problema em questão. Se elas puderem atender um domínio inteiro dos problemas, tanto melhor.

abstração mais baixo, a solução técnica do software é expressa de maneira que pode ser diretamente implementada.

À medida que diferentes níveis de abstração são alcançados, usa-se a combinação de abstrações procedurais e de dados. Uma *abstração procedural* refere-se a uma sequência de instruções que possuem uma função específica e limitada. O nome de uma abstração procedural implica sua função, porém os detalhes específicos são omitidos. Um exemplo de abstração procedural tem como nome *abrir* para uma porta. *Abrir* implica uma longa sequência de etapas procedurais (por exemplo, dirigir-se até a porta, alcançar e agarrar a maçaneta, girar a maçaneta e puxar a porta, afastar-se da porta em movimento etc.).⁶

A *abstração de dados* é um conjunto de dados com nome que descreve um objeto de dados. No contexto da abstração procedural *abrir*, podemos definir uma abstração de dados chamada **porta**. Assim como qualquer objeto de dados, a abstração de dados para **porta** englobaria um conjunto de atributos que descrevem a porta (por exemplo, tipo de porta, mudar de direção, mecanismo de abertura, peso, dimensões). Daí decorre que a abstração *abrir* faria uso de informações contidas nos atributos da abstração de dados **porta**.

8.3.2 Arquitetura

WebRef

Uma discussão mais aprofundada de arquitetura de software pode ser encontrada em www.sei.cmu.edu/ata/ata_init.html.

A *arquitetura de software* refere-se à “organização geral do software e aos modos pelos quais disponibiliza integridade conceitual para um sistema” [Sha95a]. Em sua forma mais simples, arquitetura é a estrutura ou a organização de componentes de programa (módulos), a maneira através da qual esses componentes interagem e a estrutura de dados são usadas pelos componentes. Em um sentido mais amplo, entretanto, os componentes podem ser generalizados para representar os principais elementos de um sistema e suas interações.

Uma meta do projeto de software é derivar um quadro da arquitetura de um sistema. Esse quadro representa a organização a partir da qual atividades mais detalhadas de projeto são conduzidas. Um conjunto de padrões de arquitetura permite a um engenheiro de software reusar soluções-padrão para problemas similares.

Shaw e Garlan [Sha95a] descrevem um conjunto de propriedades que devem ser especificadas como parte de um projeto de arquitetura:

Propriedades estruturais. Esse aspecto do projeto da representação da arquitetura define os componentes de um sistema (por exemplo, módulos, objetos, filtros) e a maneira pela qual os componentes são empacotados e interagem entre si. Por exemplo, objetos são empacotados para encapsularem dados e processamento que manipula esses dados e interagem por meio da chamada dos métodos.

Propriedades não funcionais. A descrição do projeto de arquitetura deve tratar a maneira pela qual o projeto da arquitetura atinge os requisitos de desempenho, capacidade, confiabilidade, segurança, adaptabilidade e outras características do sistema, que não representam as funcionalidades diretamente acessadas pelos usuários.

Famílias de sistemas. O projeto de arquitetura deve tirar proveito de padrões reusáveis comumente encontrados no projeto de famílias de sistemas similares. Em essência, o projeto deve ter a habilidade de reutilizar os componentes que fazem parte da arquitetura.

Dada a especificação dessas propriedades, o projeto da arquitetura pode ser representado usando-se um ou mais modelos diferentes [Gar95]. Os *modelos estruturais* representam a arquitetura como um conjunto organizado de componentes de programa. Esses modelos aumentam o nível de abstração do projeto tentando identificar projetos arquiteturais reusáveis encontrados em tipos de aplicações similares. Os *modelos dinâmicos* tratam dos aspectos comportamentais da arquitetura do programa, indicando como a estrutura ou configuração do sistema pode mudar em função de eventos externos. Os *modelos de processos* concentram-se no projeto do



Não deixe que a arquitetura aconteça ao acaso. Ao fazer isso, você consumirá o restante do projeto adequando-a para a forma forçada ao projeto. Projete a arquitetura explicitamente.

⁶ Deve-se notar, entretanto, que um conjunto de operações pode ser substituído por outro, desde que a função implicada pela abstração procedural permaneça a mesma. Consequentemente, as etapas necessárias para implementar *abrir* mudariam dramaticamente se a porta fosse automática e ligada a um sensor.

processo técnico ou do negócio que o sistema deve atender. Por fim, os *modelos funcionais* podem ser utilizados para representar a hierarquia funcional de um sistema.

Desenvolveu-se uma série de *linguagens de descrição de arquitetura* (*architectural description languages*, ADLs) diferente para representar esses modelos [Sha95b]. Embora tenham sido propostas diversas ADLs, a maioria fornece mecanismos para descrever componentes de sistema e a maneira através da qual estão conectados entre si.

Você deve notar que há certo debate em torno do papel da arquitetura no projeto. Alguns pesquisadores argumentam que a obtenção da arquitetura de software deve ser separada do projeto e ocorre entre as ações da engenharia de requisitos e ações de projeto mais convencionais. Outros acreditam que a obtenção da arquitetura é parte integrante do processo de projeto. A maneira pela qual a arquitetura de software é caracterizada e seu papel no projeto são discutidos no Capítulo 9.

“Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente, e então descreve o núcleo da solução para esse problema, de uma forma tal que podemos usar a solução milhões de vezes, sem jamais fazê-lo da mesma forma.”

Christopher
Alexander

8.3.3 Padrões

Brad Appleton define *padrão de projeto* da seguinte maneira: “Padrão é parte de um conhecimento consolidado já existente que transmite a essência de uma solução comprovada para um problema recorrente em certo contexto, em meio a preocupações concorrentes” [App00]. Em outras palavras, um padrão de projeto descreve uma estrutura de projeto que resolve uma particular categoria de problemas de projeto em um contexto específico e entre “forças” que direcionam a maneira pela qual o padrão é aplicado e utilizado.

O intuito de cada padrão de projeto é fornecer uma descrição que permite a um projetista determinar (1) se o padrão se aplica ou não ao trabalho em questão, (2) se o padrão pode ou não ser reutilizado (e, portanto, poupando tempo) e (3) se o padrão pode servir como um guia para desenvolver um padrão similar, mas funcional ou estruturalmente diferente. Os padrões de projeto são discutidos de forma detalhada no Capítulo 12.

8.3.4 Separação por interesses (por afinidades)

A *separação por interesses* é um conceito de projeto [Dij82] que sugere que qualquer problema complexo pode ser tratado mais facilmente se for subdividido em trechos a ser resolvidos e/ou otimizados independentemente. *Interesse* se manifesta como uma característica ou comportamento especificados como parte do modelo de requisitos do software. Por meio da separação por interesses em blocos menores e, portanto, mais administráveis, um problema toma menos tempo para ser resolvido.

Para dois problemas, p_1 e p_2 , se a complexidade percebida de p_1 for maior do que a percebida para p_2 , segue que o esforço necessário para solucionar p_1 é maior do que o esforço necessário para solucionar p_2 . Como caso geral, esse resultado é intuitivamente óbvio. Leva mais tempo para resolver um problema difícil.

Segue também que a complexidade percebida de dois problemas, quando estes são combinados, normalmente é maior do que soma da complexidade percebida quando cada um deles é tomado separadamente. Isso nos leva a uma estratégia dividir-para-conquistar — é mais fácil resolver um problema complexo quando o subdividimos em partes gerenciáveis. Isso tem implicações importantes em relação à modularidade do software.

A separação por interesses manifesta-se em outros conceitos relacionados ao projeto: modularidade, aspectos, independência funcional e refinamento.

8.3.5 Modularidade

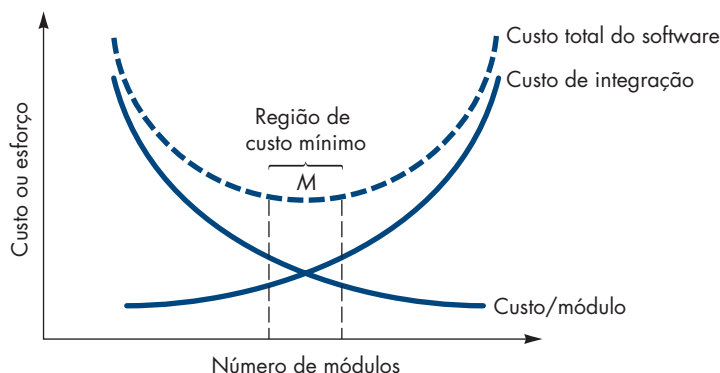
Modularidade é a manifestação mais comum da separação por interesses. O software é dividido em componentes separadamente especificados e endereçáveis, algumas vezes denominados *módulos*, que são integrados para satisfazer os requisitos de um problema.

Afirmou-se que “modularidade é o único atributo de software que possibilita que um programa seja intelectualmente gerenciável” [Mye78]. Software monolítico (um grande programa



O argumento para a separação por interesses pode se estender muito mais. Se dividirmos um problema em um número excessivo de problemas muito pequenos, resolver cada um deles será fácil, porém, juntá-los em uma solução — integração — talvez seja muito difícil.

FIGURA 8.2

Modularidade e custo do software

composto de um único módulo) não pode ser facilmente entendido por um engenheiro de software. O número de caminhos de controle, abrangência de referência, número de variáveis e complexidade geral tornaria o entendimento próximo do impossível. Em quase todos os casos, devemos dividir o projeto em vários módulos, para facilitar a compreensão e, conseqüentemente, reduzir o custo necessário para construir o software.

Recapitulando nossa discussão sobre separação por interesses, é possível concluir que, se subdividirmos o software indefinidamente, o esforço exigido para desenvolvê-lo seja pequeno! Infelizmente, outros fatores entram em jogo, invalidando essa conclusão. Referindo-se à Figura 8.2, o esforço (custo) para desenvolver um módulo de software individual realmente diminui à medida que o número total de módulos cresce. Dado o mesmo conjunto de requisitos, também com mais módulos, significa tamanho individual menor. Entretanto, à medida que o número de módulos aumenta, o esforço (custo) associado à integração dos módulos também cresce. Essas características levam a um custo total ou curva de esforço mostrada na figura. Existe um número M de módulos que resultaria em um custo de desenvolvimento mínimo, porém não temos a sofisticação suficiente para prever M com certeza.

? Qual o número exato de módulos para um dado sistema?

As curvas da Figura 8.2 nos dão uma útil orientação qualitativa quando a modularidade é considerada. Devemos modularizar, mas tomar cuidado para permanecer nas vizinhanças de M . Devemos evitar modularizar a menos ou a mais. Mas como saber a vizinhança de M ? Quanto modular devemos fazer com que um software seja? As respostas a essas perguntas exigem um entendimento de outros conceitos de projeto considerados posteriormente neste capítulo.

Modularizamos um projeto (e o programa resultante) de modo que o desenvolvimento possa ser planejado mais facilmente; incrementos de software possam ser definidos e entregues; as mudanças possam ser mais facilmente acomodadas; os testes e depuração possam ser conduzidos de forma mais eficaz e a manutenção no longo prazo possa ser realizada sem efeitos colaterais sérios.

8.3.6 Encapsulamento⁷ de informações

O conceito de modularidade nos leva a uma questão fundamental: “Como decompor uma solução de software para obter o melhor conjunto de módulos?”. O princípio de encapsulamento de informações [Par72] sugere que os módulos sejam “caracterizados por decisões de projeto que ocultem (cada uma delas) de todas as demais”. Em outras palavras, os módulos devem ser especificados e projetados de modo que as informações (algoritmos e dados) contidas em um módulo sejam inacessíveis por parte de outros módulos que não necessitam tais informações, disponibilizando apenas os itens que interessam aos outros módulos.

⁷ N. de T.: Encapsular é uma técnica de engenharia largamente usada. Por exemplo, um componente de hardware digital é projetado da seguinte forma: esconde aspectos que não interessam aos demais componentes e publica aspectos úteis aos demais componentes.

PONTO-CHAVE

O intuito de encapsulamento de informações é esconder os detalhes das estruturas de dados e de processamento procedural que estão por trás da interface de acesso a um módulo. Os detalhes não precisam ser conhecidos por usuários do módulo.

Encapsular implica que efetiva modularidade pode ser conseguida por meio da definição de um conjunto de módulos independentes que passam entre si apenas as informações necessárias para realizar determinada função do software. A abstração ajuda a definir as entidades procedurais (ou informativas) que constituem o software. Encapsulamento define e impõe restrições de acesso tanto a detalhes procedurais em um módulo quanto em qualquer estrutura de dados local usada pelo módulo [Ros75].

O uso de encapsulamento de informações como critério de projeto para sistemas modulares fornece seus maiores benefícios quando são necessárias modificações durante os testes e, posteriormente, durante a manutenção do software. Como a maioria dos detalhes procedurais e de dados são ocultos para outras partes do software, erros introduzidos inadvertidamente durante a modificação em um módulo têm menor probabilidade de se propagar para outros módulos ou locais dentro do software.

8.3.7 Independência funcional

O conceito de independência funcional é um resultado direto da separação por interesses, da modularidade e dos conceitos de abstração e encapsulamento de informações. Em artigos marcantes sobre projeto de software, Wirth [Wir71] e Parnas [Par72] tratam técnicas de refinamento que aumentam a independência entre módulos. Trabalho posterior de Stevens, Myers e Constantine [Ste74] solidificaram o conceito.

A independência funcional é atingida desenvolvendo-se módulos com função “única” e com “aversão” à interação excessiva com outros módulos. Em outras palavras, devemos projetar software de modo que cada módulo atenda um subconjunto específico de requisitos e tenha uma interface simples quando vista de outras partes da estrutura do programa. É razoável perguntar por que a independência é importante.

Software com efetiva modularidade, isto é, módulos independentes, é mais fácil de ser desenvolvido, pois a função pode ser compartimentalizada e as interfaces simplificadas (considere as consequências quando o desenvolvimento é conduzido por uma equipe). Módulos independentes são mais fáceis de ser mantidos (e testados), pois efeitos colaterais provocados por modificação no código ou projeto são limitados, a propagação de erros é reduzida e módulos reutilizáveis são possíveis. Em suma, a independência funcional é a chave para um bom projeto, e projeto é a chave para a qualidade de um software.

A independência é avaliada usando-se dois critérios qualitativos: coesão e acoplamento. A *coesão* indica a robustez funcional relativa de um módulo. O *acoplamento* indica a interdependência relativa entre os módulos.

A coesão é uma extensão natural do conceito do encapsulamento de informações descrito na Seção 8.3.6. Um módulo coeso realiza uma única tarefa, exigindo pouca interação com outros componentes em outras partes de um programa. De forma simples, um módulo coeso deve (idealmente) fazer apenas uma coisa. Embora você sempre deva tentar ao máximo obter uma alta coesão (funcionalidade única), muitas vezes é necessário e recomendável fazer com que um componente de software realize várias funções. Entretanto, componentes “esquizofrênicos” (módulos que realizam muitas funções não relacionadas) devem ser evitados caso se queira um bom projeto.

O acoplamento é uma indicação da interconexão entre os módulos em uma estrutura de software e depende da complexidade da interface entre os módulos, do ponto onde é feito o acesso a um módulo e dos dados que passam pela interface. Em projeto de software, você deve se esforçar para obter o menor grau de acoplamento possível. A conectividade simples entre módulos resulta em software mais fácil de ser compreendido e menos sujeito a “reação em cadeia” [Ste74], provocada quando ocorrem erros, em um ponto, que se propagam por todo o sistema.

? Por que devemos nos esforçar para criar módulos independentes?

PONTO-CHAVE

Coesão é uma indicação qualitativa do grau com o qual um módulo se concentra em fazer apenas uma coisa.

PONTO-CHAVE

Acoplamento é uma indicação qualitativa do grau com o qual um módulo está conectado a outros módulos e com o mundo externo.



Existe uma tendência de ir imediatamente até o último detalhe, pulando as etapas de refinamento.

Isso induz a erros e omissões e torna o projeto muito mais difícil de ser revisado. Realize o refinamento gradual.

“É difícil ler de cabo a rabo um livro sobre princípios de mágica sem, de tempos em tempos, dar uma olhada na capa para ter certeza de que não é um livro sobre projeto de software.”

Bruce Tognazzini

PONTO-CHAVE

Preocupação em comum é alguma característica do sistema que se aplica a vários requisitos diferentes.

8.3.8 Refinamento

O refinamento gradual é uma estratégia de projeto *top-down* proposta originalmente por Niklaus Wirth [Wir71]. Um programa é desenvolvido refinando-se sucessivamente níveis de detalhes procedurais. É desenvolvida uma hierarquia através da decomposição de uma declaração macroscópica da função (uma abstração procedural) de forma gradual até que as declarações da linguagem de programação sejam atingidas.

Refinamento é, na verdade, um processo de *elaboração*. Começamos com um enunciado da função (ou descrição de informações) definida em um alto nível de abstração. O enunciado descreve a função ou informações conceitualmente, mas não fornece nenhuma informação sobre o funcionamento interno da função ou da estrutura interna das informações. Em seguida, elaboramos a declaração original, fornecendo cada vez mais detalhes à medida que ocorre cada refinamento (elaboração) sucessivo.

Abstração e refinamento são conceitos complementares. A abstração nos permite especificar procedimento e dados internamente, mas suprimir a necessidade de que “estranhos” tenham conhecimento de detalhes de baixo nível. O refinamento nos ajuda a revelar detalhes menores à medida que o projeto avança. Ambos os conceitos permitem que criemos um modelo de projeto completo à medida que o projeto evolui.

8.3.9 Aspectos

À medida que ocorre análise de requisitos, vai sendo revelado um conjunto de “interesses (ou afinidades)”. Entre tais interesses “temos os requisitos, os casos de uso, as características, as estruturas de dados, questões de qualidade de serviço, variações, limites de propriedade intelectual, colaborações, padrões e contratos” [AOS07]. Idealmente, um modelo de requisitos pode ser organizado de forma que lhe permita isolar grupos de interesses (requisitos) para que possam ser considerados independentemente. Na prática, entretanto, alguns desses interesses abrangem o sistema inteiro e não pode ser facilmente dividido em compartimentos.

Quando um projeto se inicia, os requisitos são refinados em uma representação de projeto modular. Consideremos dois requisitos, *A* e *B*. O requisito *A* *intersecciona* o requisito *B* “se tiver sido escolhida uma decomposição [refinamento] de software em que *B* não pode ser satisfeito sem levar em conta *A*” [Ros04].

Consideremos, por exemplo, dois requisitos para a WebApp **CasaSeguraGarantida.com**. O requisito *A* é descrito por meio do caso de uso **AVC-EVC** discutido no Capítulo 6. O refinamento de um projeto poderia se concentrar naqueles módulos que permitiriam a um usuário registrado acessar imagens de vídeo de câmeras distribuídas em um ambiente. O requisito *B* é um requisito de segurança genérico que afirma que *um usuário registrado tem de ser validado antes de usar CasaSeguraGarantida.com*. Esse requisito se aplica a todas as funções disponíveis para os usuários registrados de *CasaSegura*. À medida que ocorre o refinamento de projeto, *A** é uma representação de projeto para o requisito *A*, e *B** é uma representação de projeto para o requisito *B*. Consequentemente, *A** e *B** são representações de interesses, e *B** *tem intersecção com A**.

Aspecto é uma representação de um interesse em comum. Consequentemente, a representação de projeto, *B**, do requisito *um usuário registrado tem de ser validado antes de poder usar CasaSeguraGarantida.com*, é um aspecto da WebApp *CasaSegura*. É importante identificar aspectos de modo que o projeto possa acomodá-los apropriadamente à medida que ocorre o refinamento e a modularização. Em um contexto ideal, um aspecto é implementado como um módulo (componente) separado, em vez de fragmentos de software que estão “espalhados” ou “emaranhados” através de vários componentes [Ban06]. Para tanto, a arquitetura de projeto deve oferecer suporte a um mecanismo para definição de aspectos — um módulo que possibilite que um interesse seja implementado e atenda aos demais interesses que ele interseccione.

WebRef

Excelentes recursos sobre refabricação podem ser encontrados em www.refactoring.com.

WebRef

Uma série de padrões de refatoração podem ser encontrados em http://c2.com/cgi/wiki?Refactoring_Patterns.

8.3.10 Refatoração

Uma importante atividade sugerida por diversos métodos ágeis (Capítulo 3), a *refatoração* é uma técnica de reorganização que simplifica o projeto (ou código) de um componente sem mudar sua função ou comportamento. Fowler [Fow00] define refatoração da seguinte maneira: “Refatoração é o processo de mudar um sistema de software de tal forma que não altere o comportamento externo do código [projeto], embora melhore sua estrutura interna”.

Quando um software é refabricado, o projeto existente é examinado em termos de redundância, elementos de projeto não utilizados, algoritmos ineficientes ou desnecessários, estruturas de dados mal construídas ou inapropriadas, ou qualquer outra falha de projeto que possa ser corrigida para produzir um projeto melhor. Por exemplo, uma primeira iteração de projeto poderia gerar um componente que apresentasse baixa coesão (realizar três funções que possuem apenas relacionamento limitado entre si). Após cuidadosa consideração, talvez decidamos que o componente devesse ser refabricado em três componentes distintos, cada um apresentando alta coesão.

O resultado será um software mais fácil de se integrar, testar e manter.

CASA SEGURA**Conceitos de projeto**

Cena: Sala do Vinod, quando começa a modelagem de projetos.

Atores: Jamie, Vinod e Ed — todos membros da equipe de engenharia de software do CasaSegura. Também participa Shakira, novo membro da equipe.

Conversa:

[Todos os quatro membros da equipe acabaram de voltar de um seminário intitulado “Aplicação de Conceitos Básicos de Projeto”, oferecido por um professor de computação.]

Vinod: Vocês tiveram algum proveito do seminário?

Ed: Já conhecia grande parte do que foi falado, mas não é uma má ideia ouvir novamente, suponho.

Jamie: Quando era aluno de Ciências da Computação, nunca realmente entendi por que o encapsulamento de informações era tão importante como diziam.

Vinod: Porque... Essencialmente... É uma técnica para reduzir a propagação de erros em um programa. Na verdade, a independência funcional também realiza a mesma coisa.

Shakira: Eu não fiz Ciências da Computação, portanto, um monte de coisas que o professor mencionou é novidade para mim. Sou capaz de gerar bom código e rapidamente. Não vejo por que isso é tão importante.

Jamie: Vi seu trabalho, Shak, e sabe de uma coisa, você já faz grande parte do que foi dito naturalmente... É por isso que seus projetos e códigos funcionam.

Shakira (sorrindo): Bem, sempre realmente tento subdividir o código, mantê-lo concentrado em algo, manter as interfaces simples e restritas, reutilizar código sempre que posso... Esse tipo de coisa.

Ed: Modularidade, independência funcional, encapsulamento, padrões... Sabe.

Jamie: Ainda me lembro do primeiro curso de programação que fiz... Eles nos ensinaram a refinar o código iterativamente.

Vinod: O mesmo pode ser aplicado ao projeto, sabe.

Vinod: Os únicos conceitos que ainda não havia ouvido falar foram “aspectos” e “refabricação”.

Shakira: Isso é usado em *Extreme Programming*, acho que foi isso que ele disse.

Ed: Exato. Não é muito diferente do refinamento, apenas que você o faz depois que o projeto ou código esteja completo. Acontece uma espécie de otimização no software, se você quer saber.

Jamie: Retornemos ao projeto CasaSegura. Imagino que devamos colocar esses conceitos em nossa lista de controle de revisão à medida que desenvolvermos o modelo de projeto para o CasaSegura.

Vinod: concordo. Mas tão importante quanto, vamos todos nos comprometer a pensar nelas enquanto desenvolvemos o projeto.

8.3.11 Conceitos de projeto orientado a objetos

O paradigma orientado a objetos (*object oriented*, OO) é largamente utilizado na engenharia de software moderna. O Apêndice 2 é dirigido àqueles que talvez não estejam familiarizados com conceitos de projeto OO como classes e objetos, herança, mensagens e polimorfismo, entre outros.

8.3.12 Classes de projeto

O modelo de requisitos define um conjunto de classes de análise (Capítulo 6). Cada um descreve algum elemento do domínio do problema, concentrando-se nos aspectos do problema

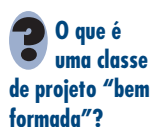
visíveis ao usuário. O nível de abstração de uma classe de análise é relativamente alto. À medida que o modelo de projeto evoluir, definiremos um conjunto de *classes de projeto* que refinem as classes de análise, fornecendo detalhes de projeto que permitirão que as classes sejam implementadas, e implementem uma infraestrutura de software que suporte a solução do negócio. Podem ser desenvolvidos cinco tipos diferentes de classes de projeto, cada um deles representando uma camada diferente da arquitetura de projeto [Amb01]:



- *Classes de interfaces do usuário* definem todas as abstrações necessárias para a interação humano-computador (*human-computer interaction*, HCI). Em muitos casos, a HCI acontece no contexto de uma *metáfora* (por exemplo, um talão de cheques, um formulário de pedidos, uma máquina de fax) e as classes de projeto para uma interface poderiam ser representações visuais dos elementos da metáfora.
- *Classes de domínio de negócio* normalmente são refinamentos das classes de análise definidas anteriormente. As classes identificam os atributos e serviços (métodos) necessários para implementar algum elemento do domínio de negócio.
- *Classes de processos* implementam as abstrações de aplicação de baixo nível necessárias para a completa gestão das classes de domínio de negócio.
- *Classes persistentes* representam repositórios de dados (por exemplo, um banco de dados) que persistirá depois da execução do software.
- *Classes de sistema* implementam funções de gerenciamento e controle de software que permitam ao sistema operar e comunicar em seu ambiente computacional e com o mundo exterior.

À medida que a arquitetura se forma, o nível de abstração é reduzido enquanto cada classe de análise é transformada em uma representação de projeto. As classes de análise representam objetos de dados (e serviços associados aplicados a eles) usando o jargão do domínio de negócio. As classes de projeto apresentam significativamente mais detalhes técnicos como um guia para a implementação.

Arlow e Neustadt [Arl02] sugerem que cada classe de projeto seja revista para garantir que seja “bem formada”. Eles definem quatro características de uma classe de projeto bem formada:



Completa e suficiente. Uma classe de projeto deve ser o encapsulamento completo de todos os atributos e métodos que podem ser razoavelmente esperados (baseado em uma interpretação inteligente do nome da classe) para a classe. Por exemplo, a classe **Cena** definida para software de edição de vídeo é completa apenas se contiver todos os atributos e métodos que podem ser razoavelmente associados com a criação de uma cena de vídeo. Suficiência garante que a classe de projeto contenha apenas os métodos suficientes para atingir o objetivo da classe, nem mais nem menos.

Primitivismo. Os métodos associados a uma classe de projeto deveriam se concentrar na realização de um serviço para a classe. Assim que o serviço tivesse sido implementado com um método, a classe não deveria realizar de outra maneira a mesma coisa. Por exemplo, a classe **VideoClipe** para um software de edição de vídeo poderia ter atributos **ponto de início** e **ponto final** para indicar os pontos de início e fim do clipe (note que uma fita virgem carregada no sistema talvez seja mais longa do que o clipe que é usado). Os métodos, *estabelecerPontoInício()* e *estabelecerPontoFinal()*, fornecem os únicos meios para estabelecer os pontos de início e fim do clipe.

Alta coesão. Uma classe de projeto coesa tem um conjunto de responsabilidades pequeno e focado e de forma resoluta aplica atributos e métodos para implementar essas responsabilidades. Por exemplo, a classe **VideoClipe** poderia conter um conjunto de métodos para editar o videoclipe. Desde que cada método se concentra exclusivamente nos atributos associados ao videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário para as classes colaborarem entre si. Entretanto, a colaboração deveria ser mantida em um mínimo aceitável. Se um modelo de

projeto for altamente acoplado (todas as classes de projeto colaboram com todas as demais classes de projeto), o sistema é difícil de implementar, testar e manter ao longo do tempo. Em geral, as classes de projeto em um subsistema deveriam ter apenas um conhecimento limitado das demais classes. Essa restrição, chamada *Lei de Demeter* [Lie03], sugere que um método deveria enviar mensagens apenas para métodos em classes vizinhas.⁸

CASA SEGURA



Refinamento de uma classe de análise em uma classe de projeto

Cena: Sala do Ed, quando começa o modelamento de projetos.

Atores: Vinod e Ed — membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

[Ed está trabalhando na classe **Planta** (veja discussão no quadro da Seção 6.5.3 e na Figura 6.10) e a refinou para o modelo de projeto.]

Ed: Então, você se lembra da classe **Planta**, certo? Ela é usada como parte das funções de vigilância e gestão da casa.

Vinod (confirmando com a cabeça): É isso mesmo, parece que nós a usamos como parte de nossas discussões CRC para gestão da casa.

Ed: Usamos. De qualquer maneira, estou refinando-a para o projeto. Gostaria de mostrar como realmente implementaremos a classe **Planta**. Minha ideia é implementá-la como um conjunto de listas ligadas [uma estrutura de dados específica] Portanto... Eu tinha de refinar a classe de análise

Planta (Figura 6.10) e, na verdade, simplificá-la.

Vinod: A classe de análise mostrava coisas apenas no domínio do problema, bem, na verdade na tela do computador, o que era visível para o usuário final, certo?

Ed: Isso, mas para a classe de projeto **Planta**, tive que acrescentar algumas coisas específicas da implementação. Precisava mostrar que **Planta** é uma agregação de segmentos — daí a classe **Segmento** — e que a classe **Segmento** é composta por listas de segmentos de parede, janelas, portas e assim por diante. A classe **Câmera** colabora com **Planta** e, obviamente, podem existir muitas câmeras na planta.

Vinod: Ufa, vejamos uma figura desta nova classe de projeto **Planta**. [Ed mostra a Vinod o desenho apresentado na Figura 8.3.]

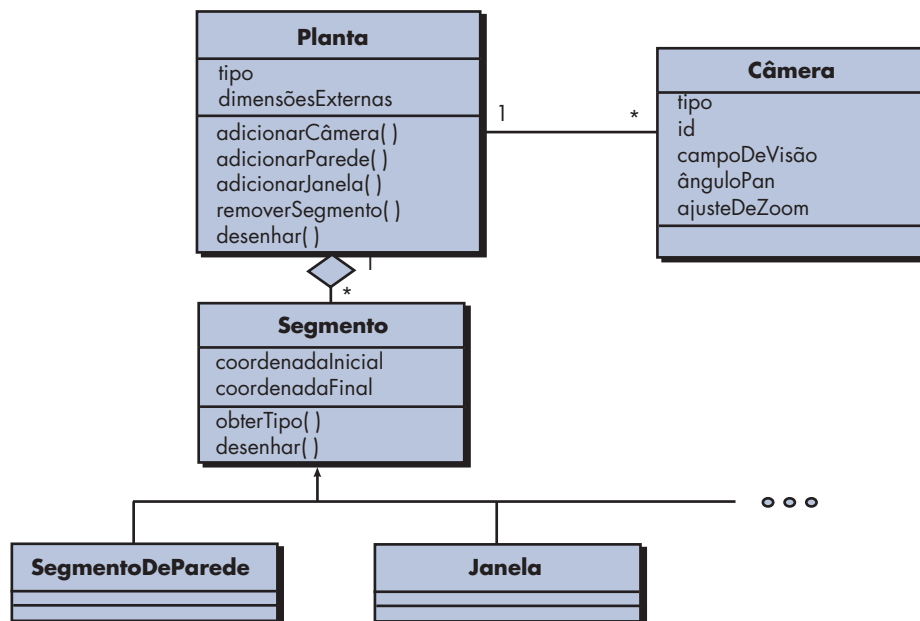
Vinod: Ok, vejo que você está tentando fazer. Isso lhe permite modificar facilmente a planta, pois novos itens podem ser acrescentados ou eliminados da lista — a agregação — sem quaisquer problemas.

Ed (meneando a cabeça): É isso aí, acho que vai funcionar.

Vinod: Eu também.

FIGURA 8.3

Classe de projeto para a **Planta** e agregação composta para a classe (veja no quadro de discussão)



⁸ Uma maneira menos formal de expressar a Lei de Demeter seria: "Cada unidade deve conversar apenas com seus amigos; não converse com estranhos".

8.4 O MODELO DE PROJETO

PONTO-CHAVE

O modelo de projeto possui quatro elementos principais: dados, arquitetura, componentes e interface.

“Questões como se o projeto é necessário ou acessível não vêm ao caso: o projeto é inevitável. A alternativa para um bom projeto é um projeto ruim, e não nenhum projeto em absoluto.”

Douglas Martin

O modelo de projeto pode ser visto em duas dimensões diferentes conforme ilustrado na Figura 8.4. A *dimensão processo* indica uma evolução do modelo de projeto à medida que as tarefas de projeto são executadas como parte do processo do software. A *dimensão da abstração* representa o nível de detalhe à medida que cada elemento do modelo de análise é transformado em um equivalente de projeto e então refinado iterativamente. Referindo-se à Figura 8.4, a linha tracejada indica o limite entre os modelos de análise e de projeto. Em alguns casos, é possível ter uma clara distinção entre os modelos de análise e de projeto. Em outros, o modelo de análise vai lentamente se misturando ao de projeto e essa distinção clara é menos evidente.

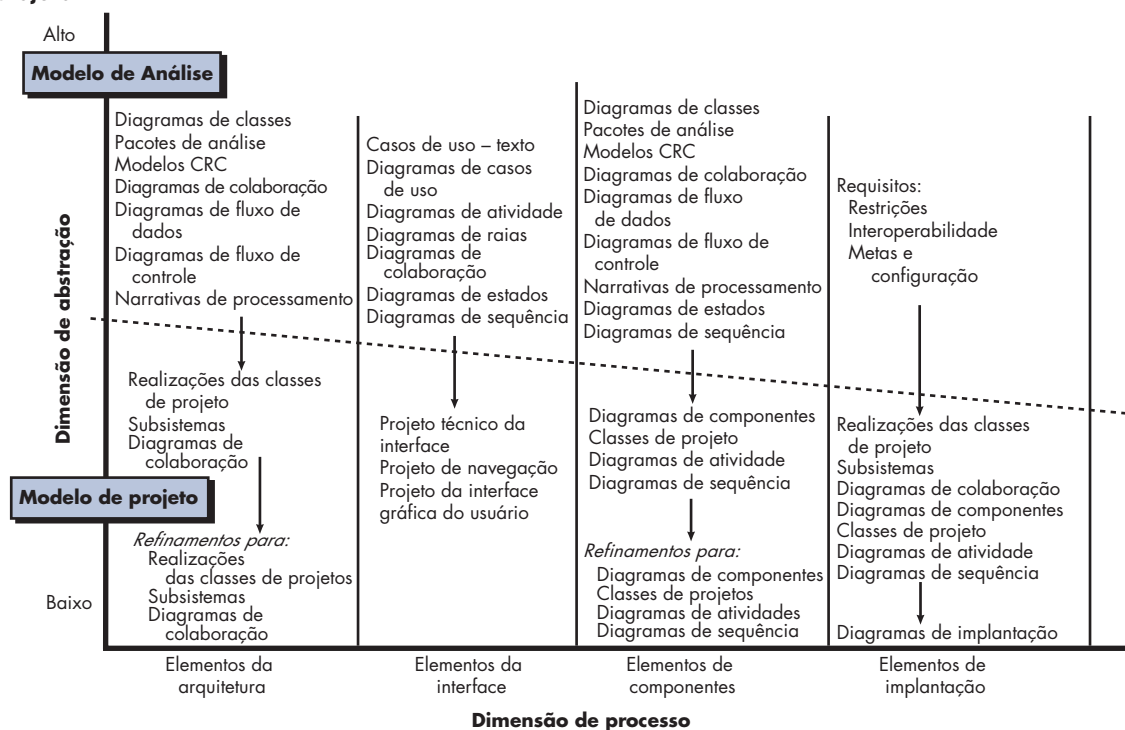
Os elementos do modelo de projeto usam vários dos diagramas⁹ UML utilizados no modelo de análise. A diferença é que esses diagramas são refinados e elaborados como parte do projeto; são fornecidos detalhes mais específicos à implementação e enfatizados a estrutura e o estilo da arquitetura, os componentes que residem nessa arquitetura, bem como as interfaces entre os componentes e com o mundo exterior.

Deve-se notar, entretanto, que os elementos de modelo indicados ao longo do eixo horizontal nem sempre são desenvolvidos de maneira sequencial. Na maioria dos casos, um projeto de arquitetura preliminar prepara o terreno e é seguido pelos projetos de interfaces e projeto de componentes, que normalmente ocorrem em paralelo. O modelo de implantação em geral é retardado até que o projeto tenha sido completamente desenvolvido.

Podemos aplicar padrões de projeto (Capítulo 12) em qualquer ponto durante o projeto. Estes possibilitam a utilização de conhecimentos adquiridos em projetos anteriores a problemas de domínios específicos encontrados e solucionados por outros.

FIGURA 8.4

Dimensões do modelo de projeto



9 O Apêndice 1 apresenta um tutorial sobre conceitos básicos e notação da UML.

PONTO-CHAVE

No nível da arquitetura (aplicação), o projeto de dados se concentra em arquivos ou bancos de dados; no nível dos componentes, o projeto de dados considera as estruturas de dados necessárias para implementar os objetos de dados locais.

“Você pode usar uma borracha enquanto ainda estiver na prancheta ou uma marreta na obra depois.”

Frank Lloyd Wright

“O público está mais acostumado com projetos ruins do que bons projetos. Ele está, de fato, condicionado a preferir projetos ruins, pois é com isso que ele convive. O novo representa uma ameaça, o antigo, um sentimento de tranquilidade.”

Paul Rand

8.4.1 Elementos de projeto de dados

Assim como ocorre com outras atividades da engenharia de software, o projeto de dados (também conhecido como *arquitetura de dados*) cria um modelo de dados e/ou informações que é representado em um nível de abstração elevado (a visão do cliente/usuário dos dados). Esse modelo é então refinado em representações cada vez mais específicas da implementação que podem ser processadas pelo sistema baseado em computador. Em muitas aplicações de software, a arquitetura dos dados terá uma profunda influência na arquitetura do software que deve processá-los.

A estrutura de dados sempre foi uma importante parte do projeto de software. No nível dos componentes de programa, o projeto das estruturas de dados e os algoritmos associados necessários para manipulá-los é essencial para a criação de aplicações de alta qualidade. No nível da aplicação, a transformação de um modelo de dados (obtido como parte da engenharia de necessidades) em um banco de dados é fundamental para atingir os objetivos de negócio de um sistema. No nível de negócio, o conjunto de informações armazenadas em bancos de dados diferentes e reorganizadas em um “depósito de dados” possibilita o *data mining* ou descoberta de conhecimento que pode ter um impacto no sucesso do negócio em si. Em qualquer caso, o projeto de dados desempenha um importante papel e é discutido com mais detalhes no Capítulo 9.

8.4.2 Elementos de projeto de arquitetura

O projeto de arquitetura para software é o equivalente à planta baixa para uma casa. A planta baixa representa a distribuição dos cômodos; seus tamanhos, formas e relacionamentos entre si e as portas e janelas que possibilitam o deslocamento para dentro e fora dos cômodos. A planta baixa nos dá uma visão geral da casa. Os elementos de projeto de arquitetura nos dão uma visão geral do software.

O modelo de arquitetura [Sha96] é obtido de três fontes: (1) informações sobre o domínio de aplicação do software a ser construído; (2) elementos específicos de modelo de requisitos como os diagramas de fluxo de dados ou as classes de análise, seus relacionamentos e colaborações para o problema em questão; e (3) a disponibilidade de estilos de arquitetura (Capítulo 9) e padrões (Capítulo 12).

O projeto dos elementos de arquitetura é normalmente representado como um conjunto de subsistemas interligados, em geral derivados dos pacotes de análise contidos no modelo de requisitos. Cada subsistema pode ter sua própria arquitetura (por exemplo, uma interface gráfica do usuário poderia ser estruturada de acordo com um estilo de arquitetura preexistente para interfaces do usuário). Técnicas para obtenção de elementos específicos do modelo de arquitetura são apresentadas no Capítulo 9.

8.4.3 Elementos de projeto de interfaces

O projeto de interfaces para software é análogo a um conjunto de desenhos detalhados (e especificações) para portas, janelas e ligações externas de uma casa. Esses desenhos representam o tamanho e a forma das portas e janelas, a maneira por meio da qual funcionam, a maneira pela qual as ligações de serviços públicos (por exemplo, água, energia elétrica, gás, telefone) entram na casa e são distribuídas entre os cômodos representados na planta. Eles nos informam onde se encontra a campainha, se deve ser usado ou não um porteiro eletrônico para anunciar a presença de um visitante e como um sistema de segurança deve ser instalado. Em essência, os desenhos detalhados (e especificações) para portas, janelas e ligações externas nos notificam como as coisas e as informações fluem para dentro e para fora da casa e no interior dos cômodos que fazem parte da planta. Os elementos de projeto de interfaces para software representam fluxos de informação que entram e saem do sistema e como são transmitidos entre os componentes definidos como parte da arquitetura.

PONTO-CHAVE

Há três partes para o elemento de projeto de interfaces: a interface do usuário, interfaces com os sistemas externos à aplicação e interfaces com componentes internos à aplicação.

“De tempos em tempos dê uma volta, relaxe um pouco, de modo que ao voltar para o trabalho seu julgamento seja mais seguro. Afaste-se um pouco; o trabalho parecerá menor e grande parte dele poderá ser vista com um pequeno exame, e a falta de harmonia e proporção serão visualizadas mais facilmente.”

Leonardo DaVinci

WebRef

Informações extremamente valiosas no projeto da UI podem ser encontradas em www.useit.com.

“Um erro comum que as pessoas cometem ao tentarem projetar algo completamente infalível é subestimar a criatividade de completos idiotas.”

Douglas Adams

Há três importantes elementos de projeto de interfaces: (1) a interface do usuário (*user interface*, UI); (2) interfaces externas para outros sistemas, dispositivos, redes ou outros produtores ou consumidores de informação; e (3) interfaces internas entre vários componentes de projeto. Esses elementos de projeto de interfaces possibilitam que o software se comunique externamente e que a comunicação interna e a colaboração entre os componentes preencham a arquitetura de software.

O projeto da UI (cada vez mais chamado *projeto de usabilidade*) é uma importante ação da engenharia de software e é considerado em detalhes no Capítulo 11. O projeto de usabilidade incorpora elementos estéticos (por exemplo, layout, cor, imagens, mecanismos de interação), elementos ergonômicos (por exemplo, o layout e a colocação de informações, metáforas, navegação da UI) e elementos técnicos (por exemplo, padrões UI, componentes reutilizáveis). Em geral, a UI é um subsistema exclusivo da arquitetura da aplicação geral.

O projeto de interfaces externas requer informações definitivas sobre a entidade para as quais as informações são enviadas ou recebidas. Em todos os casos, essas informações devem ser coletadas durante a engenharia de requisitos (Capítulo 5) e verificadas assim que o projeto de interface for iniciado.¹⁰ O projeto de interfaces externas deve incorporar a verificação de erros e (quando necessário) características de segurança apropriadas.

O projeto de interfaces internas está intimamente ligado ao projeto dos componentes (Capítulo 10). As realizações de projeto das classes de análise representam todas as operações e os esquemas de troca de mensagens necessários para habilitar a comunicação e a colaboração entre as operações em várias classes. Cada mensagem deve ser desenvolvida para acomodar a transferência de informações requeridas e os requisitos funcionais específicos da operação solicitada. Se for escolhida a abordagem clássica de entrada-processo-saída para o projeto, a interface de cada componente de software é projetada tomando como base as representações do fluxo de dados e a funcionalidade descrita em uma narrativa de processamento.

Em alguns casos, uma interface é modelada de forma bastante parecida com a de uma classe. Na UML, a interface é definida da seguinte maneira [OMG03a]: “Interface é um especificador para as operações [públicas] visíveis externamente de uma classe, componente ou outro classificador (incluindo subsistemas), sem a especificação da estrutura interna”. De maneira mais simples, interface é um conjunto de operações que descreve alguma parte do comportamento de uma classe e dá acesso a essas operações.

Por exemplo, a função de segurança do *CasaSegura* faz uso de um painel de controle que possibilita a um proprietário de imóvel controlar certos aspectos da função de segurança. Em uma versão mais avançada do sistema, as funções do painel de controle poderiam ser implementadas por meio de um PDA sem fio ou telefone celular.

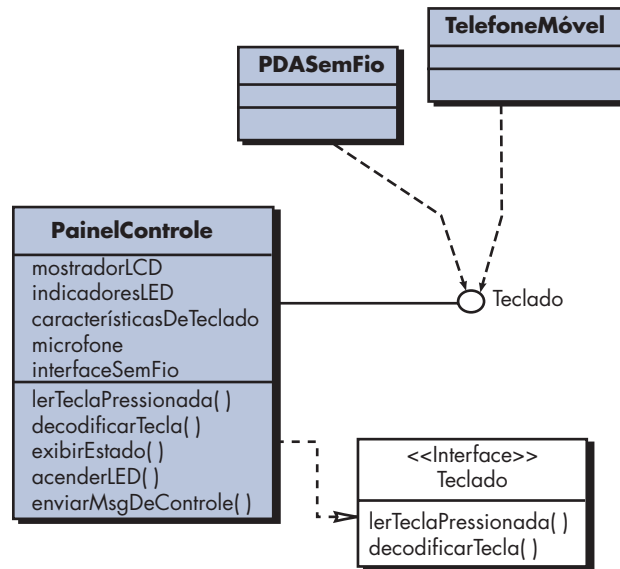
A classe **PainelControle** (Figura 8.5) fornece o comportamento associado com um teclado e, conseqüentemente, deve implementar as operações *lerTeclaPressionada()* e *decodificarTecla()*. Se essas operações tiverem de ser fornecidas para outras classes (no caso, **PDASemFio** e **TelefoneMóvel**), é útil definir uma interface conforme mostra a figura. A interface, chamada **Teclado**, é apresentada como um estereótipo <<interface>> ou como um pequeno círculo identificado ligado à classe por meio de uma linha. A interface é definida sem nenhum atributo e conjunto de operações necessários para atingir o comportamento de um teclado.

A linha tracejada com um triângulo com fundo branco em sua ponta (Figura 8.5) indica que a classe **PainelControle** fornece operações de **Teclado** como parte de seu comportamento. Na UML, isso é caracterizado como uma *realização*. Ou seja, parte do comportamento de **PainelControle** será implementada realizando as operações de **Teclado**. Essas operações serão fornecidas a outras classes que acessam a interface.

¹⁰ As características das interfaces podem mudar ao longo do tempo. Conseqüentemente, um projetista deve assegurar que a especificação para uma interface seja precisa e completa.

FIGURA 8.5

Representação da interface para PaineiControle



8.4.4 Elementos de projeto de componentes

O projeto de componentes para o software equivale a um conjunto de desenhos detalhados (e especificações) para cada cômodo de uma casa. Esses desenhos representam a fiação e o encanamento dentro de cada cômodo, a localização das tomadas e interruptores, torneiras, pias, chuveiros, banheiras, ralos, armários e banheiros. Eles também descrevem o piso a ser usado, as molduras a ser aplicadas e qualquer outro detalhe associado a um cômodo. O projeto de componentes para software descreve completamente os detalhes internos de cada componente de software. Para tanto, o projeto no nível de componente define estruturas de dados para todos os objetos de dados locais e detalhes algorítmicos para todo o processamento que ocorre em um componente e uma interface que dá acesso a todas as operações de componentes (comportamentos).

No contexto da engenharia de software orientada a objetos, um componente é representado em forma esquemática em UML conforme mostra a Figura 8.6. Nessa figura, é representado um componente chamado **GestãoDeSensor** (parte da função de segurança do *CasaSegura*). Uma seta pontilhada conecta o componente a uma classe chamada **Sensor** que é atribuída a ele. O componente **GestãoDeSensor** realiza todas as funções associadas aos sensores do *CasaSegura*, incluindo seu monitoramento e configuração. Uma discussão mais abrangente sobre diagramas de componentes é apresentada no Capítulo 10.

Os detalhes de projeto de um componente não podem ser modelados em muitos níveis de abstração diferentes. Um diagrama de atividades UML pode ser utilizado para representar processamento lógico. O fluxo procedural detalhado para um componente pode ser representado usando pseudocódigo (uma representação similar a uma linguagem de programação descrita no Capítulo 10) ou alguma outra forma esquemática (por exemplo, fluxograma ou diagrama de blocos). A estrutura algorítmica segue as regras estabelecidas para a programação estruturada (um conjunto de construções procedurais restritas). As estruturas de dados escolhidas tomando como base a natureza dos objetos de dados a ser processados normalmente são modeladas usando pseudocódigo ou a linguagem de programação para implementação.

8.4.5 Elementos de projeto de implantação

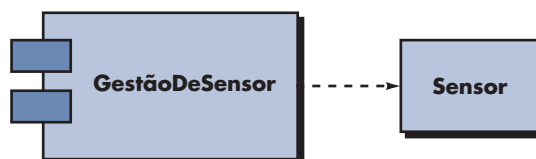
Os elementos de projeto de implantação indicam como os subsistemas e a funcionalidade de software serão alocados no ambiente computacional físico que irá suportar o software. Por

"Os detalhes não são detalhes. Eles fazem o projeto."

Charles Eames

FIGURA 8.6

Um diagrama de componentes UML



PONTO-CHAVE

Os diagramas de disponibilização começam na forma de descritores, em que o ambiente de disponibilização é descrito em termos gerais. Posteriormente, é usada a forma de instância, e os elementos da configuração são descritos explicitamente.

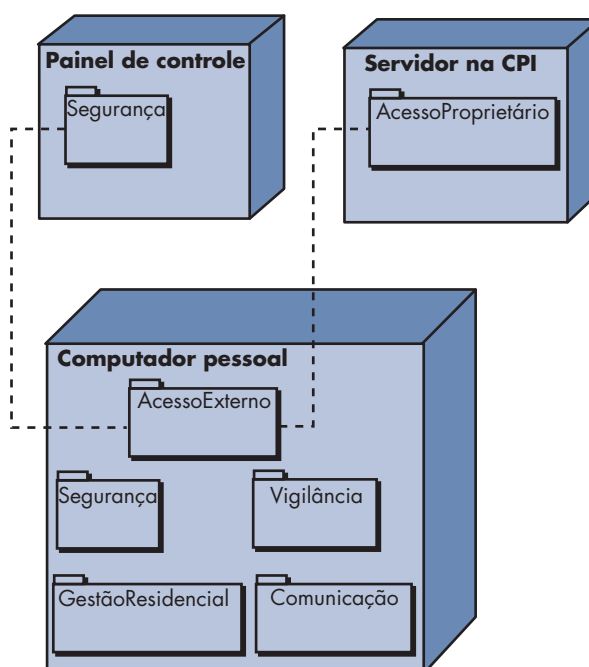
exemplo, os elementos do produto *CasaSegura* são configurados para operar dentro de três ambientes computacionais principais — um PC localizado na casa, o painel de controle *CasaSegura* e um servidor localizado na CPI Corp. (fornecendo acesso ao sistema via Internet).

Durante o projeto, um diagrama de implantação UML é desenvolvido e refinado conforme mostra a Figura 8.7. Na figura, são apresentados três ambientes computacionais (na verdade, existiriam outros com a inclusão de sensores, câmeras e outros dispositivos). Os subsistemas (funcionalidade) abrigados em cada elemento computacional são indicados. Por exemplo, o PC abriga subsistemas que implementam funções de segurança, vigilância, gestão residencial e de comunicação. Além disso, um subsistema de acesso externo foi projetado para gerenciar todas as tentativas para acessar o sistema *CasaSegura* de uma fonte externa. Cada subsistema seria elaborado para indicar os componentes que ele implementa.

O diagrama apresentado na Figura 8.7 se encontra na *forma de descritores*. Isso significa que o diagrama de disponibilização mostra o ambiente computacional, mas não indica explicitamente detalhes de configuração. Por exemplo, o “computador pessoal” não tem uma identificação adicional. Ele poderia ser um Mac ou um PC com Windows, uma estação de trabalho da Sun ou um computador com Linux. Esses detalhes são fornecidos quando o diagrama de implantação é revisitado na *forma de instância* durante os últimos estágios do projeto ou quando começa a construção. Cada instância da disponibilização (uma configuração de hardware com nome e específica) é identificada.

FIGURA 8.7

Um diagrama de implantação UML



8.5 RESUMO

O projeto de software começa quando a primeira iteração da engenharia de requisitos chega a uma conclusão. O intuito do projeto de software é aplicar um conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto de alta qualidade. O objetivo do projeto é criar um modelo de software que irá implementar corretamente todos os requisitos do cliente e trazer satisfação àqueles que o usarem. Os projetistas de software devem examinar completamente muitas alternativas de projeto e convergir para uma solução que melhor atenda às necessidades dos interessados no projeto.

O processo de projeto passa de uma visão macro do software para uma visão mais estreita que define os detalhes necessários para implementar um sistema. O processo começa concentrando-se na arquitetura. São definidos subsistemas; são estabelecidos mecanismos de comunicação entre os subsistemas; são identificados componentes; e é desenvolvida uma descrição detalhada de cada componente. Além disso, são projetadas interfaces externas, internas e para o usuário.

Os conceitos de projeto evoluíram ao longo dos primeiros 60 anos do trabalho da engenharia de software. Eles descrevem atributos de software que devem estar presentes independentemente do processo de engenharia de software escolhido, dos métodos de projeto aplicados ou das linguagens de programação usadas. Em essência, os conceitos de projeto enfatizam a necessidade da abstração como um mecanismo para a criação de componentes de software reutilizáveis; a importância da arquitetura como forma para melhor entender a estrutura geral de um sistema; os benefícios da engenharia baseada em padrões como técnica para desenvolvimento de software com capacidades já comprovadas; o valor da separação de preocupações e da modularidade eficaz como forma de tornar o software mais compreensível, mais fácil de ser testado e mantido; as consequências do encapsulamento de informações como um mecanismo para reduzir a propagação de efeitos colaterais quando da real ocorrência de erros; o impacto da independência funcional como critério para a construção de módulos eficazes; o uso do refinamento como mecanismo de projeto; a consideração de aspectos que interseccionem as necessidades do sistema; a aplicação da refabricação na otimização do projeto que é obtido; e a importância das classes orientadas a objetos e as características a elas relacionadas.

O modelo de projeto engloba quatro elementos distintos; à medida que cada um é desenvolvido, evolui uma visão mais completa do projeto. O elemento de arquitetura usa informações extraídas do domínio de aplicação, do modelo de requisitos e de catálogos disponíveis para padrões e estilos para obter uma representação estrutural completa do software, seus subsistemas e componentes. Elementos de projeto de interfaces modelam interfaces internas e externas, bem como a interface do usuário. Elementos de componentes definem cada um dos módulos (componentes) que preenchem a arquitetura. Por fim, os elementos de disponibilização alocam a arquitetura, seus componentes e as interfaces para a configuração física que abrigará o software.

PROBLEMAS E PONTOS A PONDERAR

- 8.1.** Você projeta software ao “escrever” um programa? O que torna o projeto de software diferente da codificação?
- 8.2.** Se um projeto de software não é um programa (e não é mesmo), então o que ele é?
- 8.3.** Como avaliar a qualidade de um projeto de software?
- 8.4.** Examine o conjunto de tarefas apresentado para o projeto. Em que momento a qualidade é avaliada em um conjunto de tarefas? Como se consegue isso? Como os atributos de qualidade discutidos na Seção 8.2.1 são atingidos?
- 8.5.** Dê exemplos de três abstrações de dados e as abstrações procedurais que podem ser usadas para manipulá-las.

- 8.6.** Descreva arquitetura de software com suas próprias palavras.
- 8.7.** Sugira um padrão de projeto que você encontra em uma categoria das coisas cotidianas (por exemplo, eletrônica de consumo, automóveis, aparelhos domésticos). Descreva brevemente o padrão.
- 8.8.** Descreva a separação de preocupações com suas próprias palavras. Existe um caso em que a estratégia dividir-para-conquistar não poderia ser apropriada? Como um caso desses poderia afetar o argumento da modularidade?
- 8.9.** Quando um projeto modular deve ser implementado como um software monolítico? Como isso pode ser obtido? O desempenho é a única justificativa para a implementação de software monolítico?
- 8.10.** Discuta a relação entre o conceito de encapsulamento de informações como um atributo de modularidade eficaz e o conceito da independência de módulos.
- 8.11.** Como os conceitos de acoplamento e portabilidade de software estão relacionados? Dê exemplos para apoiar sua discussão.
- 8.12.** Aplique uma “abordagem de refinamento gradual” para desenvolver três níveis diferentes de abstrações procedurais para um ou mais dos seguintes programas: (a) desenvolver um preenchedor de cheques que, dada uma quantia numérica, imprimirá a quantia por extenso como exigido no preenchimento de cheques; (b) encontrar iterativamente as raízes de uma equação transcendente; (c) desenvolver um algoritmo de cronograma de tarefas simples para um sistema operacional.
- 8.13.** Considere o software necessário para implementar um recurso completo de navegação (usando GPS) em um dispositivo de comunicação móvel portátil. Descreva duas ou três preocupações em comum que estariam presentes. Discuta como você representaria uma dessas preocupações na forma de um aspecto.
- 8.14.** “Refabricação” significa que modificamos todo o projeto iterativamente? Em caso negativo, o que significa?
- 8.15.** Descreva brevemente cada um dos quatro elementos do modelo de projeto.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Donald Norman escreveu dois livros (*The Design of Everyday Things*, Doubleday, 1990 e *The Psychology of Everyday Things*, Harpercollins, 1988) que se tornaram clássicos na literatura de projeto e uma leitura “obrigatória” para qualquer um que projete qualquer coisa que os seres humanos usam. Adams (*Conceptual Blockbusting*, 3. ed., Addison-Wesley, 1986) é o autor de um texto essencial para os projetistas que querem alargar sua maneira de pensar. Por fim, um clássico de Polya (*How to Solve It*, 2. ed., Princeton University Press, 1988) fornece um processo genérico para resolução de problemas que podem ajudar os projetistas de software ao depararem com problemas complexos.

Seguindo a mesma tradição, Winograd et al. (*Bringing Design to Software*, Addison-Wesley, 1996) discutem projetos de software que funcionam, aqueles que não funcionam e o porquê. Um livro fascinante editado por Wixon e Ramsey (*Field Methods Casebook for Software Design*, Wiley, 1996) sugere métodos de pesquisa de campo (muito parecido com aqueles usados por antropólogos) para entender como os usuários finais realizam o trabalho deles e depois projetam software que atenda suas necessidades. Beyer e Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) oferece uma outra visão do projeto de software que integra o cliente/usuário em todos os aspectos do processo de projeto de software. Bain (*Emergent Design*, Addison-Wesley, 2008) acopla padrões, refabricação e desenvolvimento dirigido por testes em uma abordagem de projeto eficaz.

Um tratado completo de projeto no contexto da engenharia de software é apresentado por Fox (*Introduction to Software Engineering Design*, Addison-Wesley, 2006) e Zhu (*Software Design Methodology*, Butterworth-Heinemann, 2005). McConnell (*Code Complete*, 2. ed.,

Microsoft Press, 2004) traz uma excelente discussão dos aspectos práticos para projetar software de alta qualidade. Robertson (*Simple Program Design*, 3. ed., Boyd and Fraser Publishing, 1999) apresenta uma discussão introdutória do projeto de software que é útil para aqueles que estão iniciando seus estudos do assunto. Budgen (*Software Design*, 2. ed., Addison-Wesley, 2004) introduz uma série de métodos de projeto populares, comparando e contrastando cada um deles. Fowler e seus colegas (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) discutem técnicas para a otimização incremental de projetos de software. Rosenberg e Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) abordam o desenvolvimento de projetos orientados a objetos usando casos de uso como base.

Uma excelente pesquisa histórica de projeto de software está contida em uma antologia editada por Freeman e Wasserman (*Software Design Techniques*, 4. ed., IEEE, 1983). Esse tutorial reapresenta diversos artigos clássicos que formaram a base para as tendências correntes em projeto de software. Medidas da qualidade de projeto, apresentadas tanto sob as perspectivas técnica como de gerenciamento, são consideradas por Card e Glass (*Measuring Software Design Quality*, Prentice-Hall, 1990).

Uma ampla gama de fontes de informação sobre projeto de software se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevante para o projeto de software, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

**CONCEITOS-
-CHAVE**

arquétipos	241
arquitetura	230
alternativas	245
centralizada em dados	235
complexidade	247
componentes	243
em camadas	238
estilos	234
fluxo de dados	236
gêneros	232
orientada a objetos	230
padrões	238
projeto	239
refinamento	242
templates	233
ATAM	245
fabricação	268
instância	243
linguagem de descrição de arquitetura	247
mapeamento	248

Projeto foi descrito como um processo em várias etapas em que as representações de dados e a estrutura do programa, as características de interfaces e os detalhes procedurais são sintetizados com base nos requisitos de informação. Essa descrição é estendida por Freeman [Fre80]:

[P]rojeto é uma atividade que trata da tomada de decisões importantes, frequentemente de natureza estrutural. Ele divide com a programação a responsabilidade em abstrair a representação de informação e de sequências de processamento, porém o nível de detalhes é bastante diverso nos extremos. Um projeto constrói representações de programas coerentes e bem planejados que se concentram nas inter-relações das partes em um alto nível e nas operações lógicas envolvidas em níveis mais baixos.

Conforme citado no Capítulo 8, o projeto é dirigido por informações. Os métodos de projeto de software são obtidos considerando-se cada um dos três domínios do modelo de análise. Os domínios de dados, funcional e comportamental servem de orientação para a criação do projeto de software.

Neste capítulo serão apresentados os métodos necessários para criar “representações coerentes e bem planejadas” das camadas de dados e da arquitetura do modelo de projeto. O objetivo é fornecer uma abordagem sistemática para a obtenção do projeto da arquitetura — o esquema preliminar por meio do qual o software é construído.

PANORAMA

O que é? O projeto da arquitetura representa a estrutura de dados e os componentes de programa necessários para construir um sistema computacional. Ele considera o estilo de arquitetura que o sistema assumirá, a estrutura e as propriedades dos componentes que constituem o sistema, bem como as inter-relações que ocorrem entre todos os componentes da arquitetura de um sistema.

Quem realiza? Embora um engenheiro de software possa projetar tanto os dados quanto a arquitetura, essa tarefa frequentemente é distribuída a especialistas ao se construírem sistemas grandes e complexos. Um projetista de bancos de dados ou de depósito de dados cria a arquitetura de dados para um sistema. O “arquiteto de sistemas” escolhe um estilo de arquitetura apropriado com base nos requisitos obtidos durante a análise de requisitos.

Por que é importante? Você não tentaria construir uma casa sem uma planta, não é mesmo? Você também não desenharia as plantas começando pela distribuição dos encanamentos da casa. Deve-se partir do contexto geral — a casa em

si — antes de se preocupar com os detalhes. É exatamente isso o que faz o projeto da arquitetura — ele dá uma visão geral e garante que você o entendeu de forma correta.

Quais são as etapas envolvidas? O projeto da arquitetura começa pelo projeto de dados e então prossegue para a derivação de uma ou mais representações da estrutura da arquitetura do sistema. São analisados estilos ou padrões de arquitetura alternativos para obter uma estrutura mais adequada aos requisitos do cliente e atributos de qualidade. Uma vez que tenha se escolhido uma alternativa, a arquitetura é elaborada usando-se um método de projeto da arquitetura.

Qual é o artefato? Durante o projeto da arquitetura é criado um modelo de arquitetura que engloba a arquitetura de dados e a estrutura de programas. Além disso, são descritas as propriedades e as relações (interações) entre os componentes.

Como garantir que o trabalho foi realizado corretamente? A cada estágio, são revisados os produtos resultantes do projeto de software em termos de clareza, correção, completude e consistência com os requisitos e entre si.

9.1 ARQUITETURA DE SOFTWARE

Em seu famoso livro sobre o tema, Shaw e Garlan [Sha96] discutem a arquitetura de software da seguinte maneira:

Desde que o primeiro programa foi dividido em módulos, os sistemas de software passaram a ter arquiteturas e os programadores passaram a ser responsáveis pelas interações entre os módulos e as propriedades globais do conjunto. Historicamente, as arquiteturas têm sido implícitas — acidentes de implementação ou sistemas legados do passado. Os bons desenvolvedores de software muitas vezes adotaram um ou vários padrões de arquitetura como estratégias para organização de sistemas, porém usam tais padrões informalmente e não possuem nenhum meio para torná-los explícitos no sistema resultante.

Hoje, arquitetura de software efetiva e sua representação explícita tornaram-se temas dominantes em engenharia de software.

9.1.1 O que é arquitetura?

Ao se considerar a arquitetura de um edifício, vários atributos diferentes vêm à mente. No nível mais simplista, pensamos na forma geral da estrutura física. Mas, na realidade, arquitetura é muito mais do que isso. Ela é a maneira pela qual os vários componentes do edifício são integrados para formar um todo coeso. É o modo pelo qual o edifício se ajusta em seu ambiente e integra com outros edifícios da vizinhança. É o grau com que o edifício atende seu propósito expresso e satisfaz às necessidades de seu proprietário. É o sentido estético da estrutura — o impacto visual do edifício — e a maneira pela qual as texturas, cores e materiais são combinados para criar a fachada e o “ambiente de moradia”. Ela engloba também os detalhes — o projeto dos dispositivos de iluminação, o tipo de piso, o posicionamento de painéis, enfim, a lista é interminável. E, finalmente, ela é arte. Mas arquitetura também é algo mais. É constituída por “milhares de decisões, tanto as grandes como as pequenas” [Tyr05]. Algumas dessas decisões são tomadas logo no início do projeto e podem ter um impacto profundo sobre todas as ações subsequentes. Outras são postergadas ao máximo, eliminando, portanto, restrições demasiadas que levariam a uma implementação inadequada do estilo da arquitetura. Mas o que dizer da arquitetura de software? Bass, Clements e Kazman [Bas03] definem esse termo difícil de descrever da seguinte maneira:

A arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas do sistema, que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.

A arquitetura não é o software operacional, mas sim, uma representação que nos permite (1) analisar a efetividade do projeto no atendimento dos requisitos declarados, (2) considerar alternativas de arquitetura em um estágio quando realizar mudanças de projeto ainda é relativamente fácil e (3) reduzir os riscos associados à construção do software.

Essa definição enfatiza o papel dos “componentes de software” em qualquer representação de arquitetura. No contexto de projeto da arquitetura, um componente de software pode ser algo tão simples quanto um módulo de programa ou uma classe orientada a objetos, porém ele também pode ser estendido para abranger bancos de dados e “middleware” que possibilita a configuração de uma rede de clientes e servidores. As propriedades dos componentes são aquelas características necessárias para o entendimento de como eles interagem com outros componentes. No nível da arquitetura, propriedades internas (por exemplo, detalhes de um algoritmo) não são especificadas. As relações entre componentes podem ser tão simples quanto a chamada procedural de um módulo a outro ou tão complexo quanto um protocolo de acesso a banco de dados.

Alguns membros da comunidade da engenharia de software (por exemplo, [Kaz03]) fazem uma distinção entre as ações associadas à obtenção de uma arquitetura de software (aquilo que denomino “projeto da arquitetura”) e as ações aplicadas para obter o projeto de software. Conforme observado por um dos revisores desta edição:

“A arquitetura de um sistema constitui uma definição abrangente que descreve sua forma e estrutura — seus componentes e como eles se integram.”

Jerrold Grochow

PONTO-CHAVE

A arquitetura de software deve modelar a estrutura de um sistema, bem como a maneira por meio da qual dados e componentes procedurais colaboram entre si.

“Case-se depressa com sua arquitetura, arrependa-se quando quiser.”

Barry Boehm

Há uma distinta diferença entre os termos arquitetura e projeto. *Projeto* é uma instância de uma *arquitetura* da mesma forma que um objeto é instância de uma classe. Consideremos, por exemplo, a arquitetura cliente/servidor. Podemos projetar um sistema de software centralizado em redes de várias formas diferentes por meio dessa arquitetura usando a plataforma Java (Java EE) ou a plataforma Microsoft (.NET framework). Existe uma arquitetura, porém podem ser criados vários projetos baseados nessa arquitetura. Consequentemente, não podemos confundir “arquitetura” com “projeto”.

WebRef

Links úteis para diversos sites sobre arquitetura de software podem ser encontrados em www2.umassd.edu/SECenter/SAResources.html.

Embora concorde que um projeto de software seja uma instância de uma arquitetura de software específica, os elementos e estruturas definidos como parte de uma arquitetura são a raiz de qualquer projeto que evolui a partir deles. Um projeto se inicia com uma consideração de arquitetura.

Neste livro o projeto de arquitetura de software considera dois níveis da pirâmide de projeto (Figura 8.1) — projeto de dados e projeto da arquitetura. No contexto da discussão anterior, o projeto de dados permite que representemos o componente de dados da arquitetura em sistemas convencionais e as definições de classes (englobando atributos e operações), em sistemas orientados a objetos. O projeto da arquitetura concentra-se na representação da estrutura dos componentes de software, suas propriedades e interações.

9.1.2 Por que a arquitetura é importante?

Em um livro dedicado à arquitetura de software, Bass e seus colegas [Bas03] identificaram três razões-chave da importância da arquitetura de software:

- As representações da arquitetura de software são um facilitador para a comunicação entre todas as partes interessadas no desenvolvimento de um sistema computacional.
- A arquitetura evidencia decisões de projeto iniciais que terão profundo impacto em todo o trabalho de engenharia de software que vem a seguir e, tão importante quanto, no sucesso final do sistema como uma entidade operacional.
- A arquitetura “constitui um modelo relativamente pequeno e intelectualmente compreensível de como o sistema é estruturado e como seus componentes trabalham em conjunto” [Bas03].

O modelo de projeto da arquitetura e os padrões de arquitetura nele contidos são transferíveis. Gêneros, estilos e padrões de arquitetura (Seções 9.2 a 9.4) podem ser aplicados ao projeto de outros sistemas e representam um conjunto de abstrações que permitem aos engenheiros de software descrever arquitetura de modo previsível.

9.1.3 Descrições de arquitetura

Cada um de nós possui uma imagem mental daquilo que a palavra *arquitetura* significa. Entretanto, na realidade, seu significado é diverso para diferentes pessoas. A implicação é que os diferentes interessados verão uma arquitetura sob diversos pontos de vista orientados por conjuntos de interesses distintos. Isso implica que uma descrição de arquitetura é, na verdade, um conjunto de artefatos que refletem diferentes visões do sistema.

Por exemplo, o arquiteto de um importante conjunto comercial tem de trabalhar com uma série de interessados diferentes. O principal interesse do proprietário do conjunto comercial (um dos interessados) é garantir que ele seja esteticamente agradável e que forneça espaço e infraestrutura suficientes para garantir sua lucratividade. Consequentemente, o arquiteto tem de criar uma descrição usando visões de edifício que atendam os interesses do proprietário. Os pontos de vista usados são desenhos tridimensionais do edifício (para ilustrar a visão estética) e um conjunto de plantas bidimensionais para atender à necessidade do interessado em termos de espaço e infraestrutura do conjunto comercial.

Porém, o conjunto comercial tem muitos outros interessados, até mesmo o fabricante de aço de construção especial que fornecerá aço para a estrutura do prédio. Esse fabricante precisa de informações de arquitetura detalhadas sobre o aço de construção que irá su-

“A arquitetura é extremamente importante para ser deixada a cargo de uma única pessoa, independentemente de quão brilhante ela possa ser.”

Scott Ambler

PONTO-CHAVE

O modelo de arquitetura fornece uma visão gestáltica do sistema, permitindo ao engenheiro de software examiná-lo como um todo.



Seu esforço deve focalizar as representações de arquitetura que irão orientar todos os demais aspectos do projeto. Invista o tempo necessário para rever cuidadosamente a arquitetura. Um erro nessa fase terá um impacto negativo no longo prazo.

portar o edifício, incluindo tipos de vigas duplo T, suas dimensões, conectividade, materiais e muitos outros detalhes. Tais interesses são atendidos por diferentes artefatos que representam diferentes visões da arquitetura. Desenhos especializados (um outro ponto de vista) da estrutura de aço de construção do edifício focalizam apenas uma das várias preocupações do fabricante.

A descrição de arquitetura de um sistema baseado em software deve apresentar características análogas às aquelas citadas para o conjunto comercial. Tyree e Akerman [Tyr05] citam isso ao escreverem: “Os desenvolvedores desejam orientação clara e determinada sobre como prosseguir com um projeto. Os clientes querem um entendimento claro sobre as mudanças que devem ocorrer no ambiente e garantias de que a arquitetura atenderá suas necessidades de negócio. Outros arquitetos buscam um entendimento claro dos aspectos mais importantes da arquitetura”. Cada um desses “desejos” se reflete em uma visão diferente representada sob um ponto de vista diferente.

A IEEE Computer Society propôs o IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], com os seguintes objetivos: (1) estabelecer uma definição conceitual e um vocabulário para uso durante o projeto da arquitetura de software, (2) fornecer diretrizes detalhadas para representar uma descrição de arquitetura e (3) encorajar práticas de projeto de arquitetura consistentes.

O padrão IEEE define uma *descrição da arquitetura* (*architectural description, AD*) como “um conjunto de produtos para documentar uma arquitetura”. Uma descrição por si só é representada usando-se várias visões; cada *visão* é “a representação de um sistema como um todo segundo a perspectiva de um conjunto de necessidades [dos interessados] relacionado”. A *visão* é criada de acordo com regras e convenções definidas em um ponto de vista — “uma especificação das convenções para a construção e uso de uma visão” [IEE00]. Uma série de artefatos diferentes utilizados para desenvolver diferentes visões da arquitetura de software é discutida posteriormente, ainda neste capítulo.

9.1.4 Decisões de arquitetura

Cada visão desenvolvida como parte da descrição arquitetural trata uma necessidade específica do interessado. Para desenvolver cada visão (e a descrição arquitetural como um todo), o arquiteto de sistemas considera uma variedade de alternativas e, por fim, decide sobre as características de uma arquitetura específica que melhor atendam à necessidade. Consequentemente, as próprias decisões de arquitetura podem ser consideradas uma visão de arquitetura. As razões pelas quais as decisões foram tomadas fornecem uma visão sobre a estrutura de um sistema e sua adequação às necessidades dos interessados.

Como arquiteto de sistemas, podemos usar o *template* sugerido no quadro para documentar cada decisão importante. Desse modo, fornecemos os fundamentos para nosso trabalho e estabelecemos um registro histórico que pode ser útil quando mudanças de projeto tiverem de ser feitas.

9.2 GÊNEROS DE ARQUITETURA

PONTO-CHAVE

Há uma série de estilos de arquitetura diferentes que poderiam ser aplicados a um gênero específico (também denominado domínio de aplicação).

Embora os princípios fundamentais do projeto da arquitetura se apliquem a todos os tipos de arquitetura, o *gênero* normalmente ditará a abordagem de arquitetura específica para a estrutura que deve ser construída. No contexto de projeto da arquitetura, *gênero* implica uma categoria específica no domínio de software geral. Em cada categoria, pode-se encontrar uma série de subcategorias. Por exemplo, dentro do gênero *edifícios*, poderíamos encontrar os seguintes estilos gerais: casas, condomínios, prédios de apartamentos, conjuntos comerciais, prédios industriais, armazéns e assim por diante. Em cada estilo geral, poderiam ser aplicados estilos mais específicos (Seção 9.3). Cada estilo teria uma estrutura que pode ser descrita usando-se um conjunto de padrões previsíveis.

INFORMAÇÕES

**Template de descrição de decisões de arquitetura**

Cada importante decisão de arquitetura pode ser documentada para posterior revisão pelos interessados que querem entender a descrição da arquitetura proposta. O template apresentado neste quadro é uma versão resumida e adaptada de um gabarito proposto por Tyree e Ackerman [Tyr05].

Problema de projeto:	Descreva os problemas de projeto da arquitetura que devem ser tratados.
Resolução:	Informe a abordagem escolhida para tratar o problema de projeto.
Categoria:	Especifique a categoria de projeto que o problema e a resolução tratam (por exemplo, projeto de dados, estrutura de conteúdo, estrutura de componentes, integração, apresentação).
Hipóteses:	Indique quaisquer hipóteses que ajudem a dar forma à decisão.
Restrições:	Especifique quaisquer restrições do ambiente que auxiliaram a dar forma à decisão (por exemplo, padrões de tecnologia, padrões disponíveis, questões relacionadas ao projeto).

Alternativas:

Descreva brevemente as alternativas de projeto de arquitetura consideradas e por que foram rejeitadas.

Argumento:

Afirme por que escolheu a decisão em relação a outras alternativas.

Implicações:

Indique as consequências de projeto ao tomar a decisão. Como a resolução afeta outras questões do projeto da arquitetura? A resolução irá restringir o projeto de alguma forma?

Decisões relacionadas:

Que outras decisões documentadas estão relacionadas com essa decisão?

Necessidades relacionadas:

Que outros requisitos estão relacionados com essa decisão?

Artefatos:

Indique onde essa decisão irá se refletir na descrição da arquitetura.

Notas:

Faça referência a quaisquer observações feitas pela equipe ou outra documentação utilizada para tomar a decisão.

Em seu manual em constante evolução, *Handbook of Software Architecture* [Boo08], Grady Booch sugere os seguintes gêneros de arquitetura para sistemas baseados em software:

- **Inteligência artificial** — Sistemas que simulam ou ampliam a cognição, locomoção ou outros processos orgânicos humanos.
- **Comercial e sem fins lucrativos** — Sistemas que são fundamentais para a operação de um empreendimento comercial.
- **Comunicações** — Sistemas que fornecem a infraestrutura para a transferência e o gerenciamento de dados, para conectar usuários desses dados ou para apresentar dados na periferia de uma infraestrutura.
- **Autoria de conteúdo** — Sistemas que são utilizados para criar ou manipular artefatos de texto ou multimídia.
- **Dispositivos** — Sistemas que interagem com o mundo físico para fornecer algum serviço para um indivíduo.
- **Esportes e entretenimento** — Sistemas que gerenciam eventos públicos ou que geram uma experiência de entretenimento a um grande grupo.
- **Financeiros** — Sistemas que fornecem a infraestrutura para transferir e administrar dinheiro e outros valores.
- **Jogos** — Sistemas que geram uma experiência de entretenimento para indivíduos ou grupos.
- **Governo** — Sistemas que dão apoio à conduta e operações de uma entidade municipal, estadual, federal, internacional ou outras entidades políticas.
- **Industriais** — Sistemas que simulam ou controlam processos físicos.
- **Legais** — Sistemas que dão apoio ao setor judiciário.
- **Médicos** — Sistemas que diagnosticam ou curam ou contribuem para a pesquisa médica.
- **Militar** — Sistemas para consultas, comunicações, comando, controle e inteligência (C4I), bem como armamento de ataque e defesa.

“Programar sem uma arquitetura ou projeto geral em mente é como explorar uma caverna portando apenas uma lanterna: não sabemos onde estivemos, não sabemos aonde estamos indo e mal sabemos onde estamos no momento.”

Danny Thorpe

- **Sistemas operacionais** — Sistemas que se situam logo acima do hardware para fornecer serviços básicos de software.
- **Plataformas** — Sistemas que se posicionam logo acima dos sistemas operacionais para fornecer serviços avançados.
- **Científicos** — Sistemas que são utilizados para pesquisa e aplicações científicas.
- **Ferramentas** — Sistemas que são utilizados para desenvolver outros sistemas.
- **Transportes** — Sistemas que controlam veículos de navegação, terrestres, aéreos ou espaciais.
- **Serviços públicos** — Sistemas que interagem com outros softwares para fornecer algum serviço.

Sob o ponto de vista do projeto da arquitetura, cada gênero representa um desafio único. Consideremos, por exemplo, a arquitetura de software para um sistema de jogos eletrônicos. Os sistemas de jogos, algumas vezes chamados *aplicações interativas de imersão*, requerem o cálculo de algoritmos intensivos, computação gráfica sofisticada, fontes de dados multimídia *streaming*, interatividade em tempo real via entradas convencionais e não convencionais e uma série de outras necessidades especializadas.

Alexandre Francois [Fra03] sugere uma arquitetura de software para *Immersipresence*¹ que pode ser aplicada a um ambiente de jogos eletrônicos. Ele descreve a arquitetura da seguinte maneira:

SAI (*Software Architecture for Immersipresence*) é um novo modelo de arquitetura de software para projetar, analisar e implementar aplicações realizando processamento paralelo assíncrono distribuído de fluxos de dados genéricos. O objetivo do SAI é fornecer uma estrutura universal para a implementação distribuída de algoritmos e sua fácil integração em sistemas complexos... O modelo de dados subjacente e extensível, bem como o modelo de processamento paralelo assíncrono distribuído (repositório compartilhado e passagem de mensagens) híbrido, possibilita a manipulação natural e eficiente de fluxos de dados genéricos, usando-se bibliotecas ou código nativo indiferentemente. A modularidade do estilo facilita o desenvolvimento, testes e reutilização de código distribuído, bem como o rápido desenvolvimento, integração, manutenção e evolução do sistema.

Uma discussão detalhada do SAI está fora do escopo deste livro. Entretanto, é importante reconhecer que o gênero de sistemas para jogos pode ser tratado com um estilo de arquitetura (Seção 9.3) especificamente desenhado para lidar com as questões de sistemas para jogos eletrônicos. Caso tenha maior interesse, veja [Fra03].

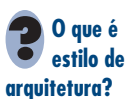
9.3 ESTILOS DE ARQUITETURA

“Por trás da mente de qualquer artista existe um padrão ou tipo de arquitetura.”

G. K. Chesterton

Quando um construtor usa a frase “colonial americano com hall central” para descrever uma casa, a maioria das pessoas familiarizadas com casas nos Estados Unidos será capaz de evocar uma imagem geral de como se parecerá a casa e como provavelmente será a sua planta. O construtor usou um *estilo arquitetônico* como um mecanismo descritivo para diferenciar a casa de outros estilos (por exemplo, casa pré-fabricada sobre uma estrutura de madeira em forma de A, rancho montado sobre uma base elevada, *Cape Cod*). Porém, mais importante ainda, o estilo arquitetônico também é um template para construção. Devem ser definidos mais detalhes da casa, suas dimensões finais especificadas, características personalizadas poderiam ser acrescentadas, também devem ser determinados os materiais de construção, o estilo — uma casa “colonial americano com hall central” — orienta o construtor em seu trabalho.

O software que é criado para sistemas computacionais também apresenta um de muitos estilos de arquitetura. Cada estilo descreve uma categoria de sistema que engloba (1) um conjunto de componentes (por exemplo, um banco de dados, módulos computacionais) que realiza uma função exigida por um sistema; (2) um conjunto de conectores que habilitam a “comunicação, coordenação e cooperação” entre os componentes; (3) restrições que definem como os



1 Francois utiliza o termo *immersipresence* para aplicações interativas e de imersão.

WebRef

Estilos de arquitetura baseados em atributos (sigla ABAS) podem ser utilizados como blocos básicos para as arquiteturas de software. Informações sobre isso podem ser obtidas em www.sei.cmu.edu/architecture/abas.html.

componentes podem ser integrados para formar o sistema; e (4) modelos semânticos que permitem a um projetista compreender as propriedades gerais de um sistema por meio da análise das propriedades conhecidas de suas partes constituintes [Bas03].

Um estilo arquitetural é uma transformação imposta no projeto do sistema inteiro. O objetivo é estabelecer uma estrutura para todos os componentes do sistema. No caso em que uma arquitetura existente deve sofrer um processo de reengenharia (Capítulo 29), a imposição de um estilo de arquitetura resultará em mudanças fundamentais na estrutura do software, incluindo uma nova atribuição da funcionalidade dos componentes [Bos00].

Um padrão de arquitetura, assim como um estilo arquitetural, impõe transformação no projeto de arquitetura. Entretanto, padrão difere de estilo em alguns modos fundamentais: (1) o escopo de um padrão é menos abrangente, concentrando-se em um aspecto da arquitetura e não na arquitetura em sua totalidade; (2) um padrão impõe uma regra sobre a arquitetura, descrevendo como o software irá tratar algum aspecto de sua funcionalidade em termos de infraestrutura (por exemplo, concorrência) [Bos00]; (3) os padrões de arquitetura (Seção 9.4) tendem a tratar questões comportamentais específicas no contexto da arquitetura (por exemplo, como as aplicações em tempo real tratam a sincronização ou as interrupções). Os padrões podem ser usados com um estilo de arquitetura para dar forma à estrutura global de um sistema. Na Seção 9.3.1, consideraremos os padrões e estilos de arquitetura e padrões mais comumente utilizados em software.



Estruturas de arquitetura canônicas

Em essência, a arquitetura de software representa uma estrutura em que algum conjunto de entidades (normalmente denominadas *componentes*) é interligado por um conjunto de relacionamentos definidos (normalmente denominados *conectores*). Tanto os componentes quanto os conectores estão associados a um conjunto de *propriedades* que permitem ao projetista diferenciar os tipos de componentes e conectores que podem ser usados. Mas que tipos de estruturas (componentes, conectores e propriedades) podem ser usados para descrever uma arquitetura? Bass e Kazman [Bas03] sugerem cinco estruturas de arquitetura fundamentais ou canônicas:

Estrutura funcional. Os componentes representam entidades funcionais ou de processamento. Os conectores representam interfaces que fornecem a capacidade de “usar” ou “passar dados para” um componente. As propriedades descrevem a natureza dos componentes e a organização das interfaces.

Estrutura de implementação. “Os componentes podem ser pacotes, classes, objetos, procedimentos, funções, métodos etc., todos os quais são veículos para empacotar funcionalidade em vários níveis de abstração” [Bas03]. Os conectores incluem a habilidade de passar dados e controle, compartilhar dados, “uso” e “instância-de”. As propriedades concentram-se nas características de qualidade (por exemplo, facilidade de

manutenção, reusabilidade) resultantes quando é implementada uma estrutura.

Estrutura de concorrência. Os componentes representam “unidades de concorrência” organizadas como tarefas paralelas ou *threads*. “As relações [conectores] incluem sincronizações-com, é-de maior prioridade-que, envia-dados-a, não é possível-executar-sem e não é possível-executar-com. Entre as propriedades relevantes a essa estrutura temos prioridade, preempção e tempo de execução” [Bas03].

Estrutura física. Essa estrutura é similar ao modelo de implantação desenvolvido como parte do projeto. Os componentes são o hardware físico onde reside o software. Os conectores são as interfaces entre os componentes de hardware e as propriedades tratam a capacidade, largura de banda, desempenho e outros atributos.

Estrutura de desenvolvimento. Essa estrutura define os componentes, artefatos e outras fontes de informação necessárias à medida que a engenharia de software prossegue. Os conectores representam os relacionamentos entre os artefatos, e as propriedades identificam as características de cada item. Cada uma dessas estruturas apresenta visão diferente da arquitetura de software, expondo informações úteis para a equipe de software à medida que a modelagem e a construção prosseguem.

INFORMAÇÕES

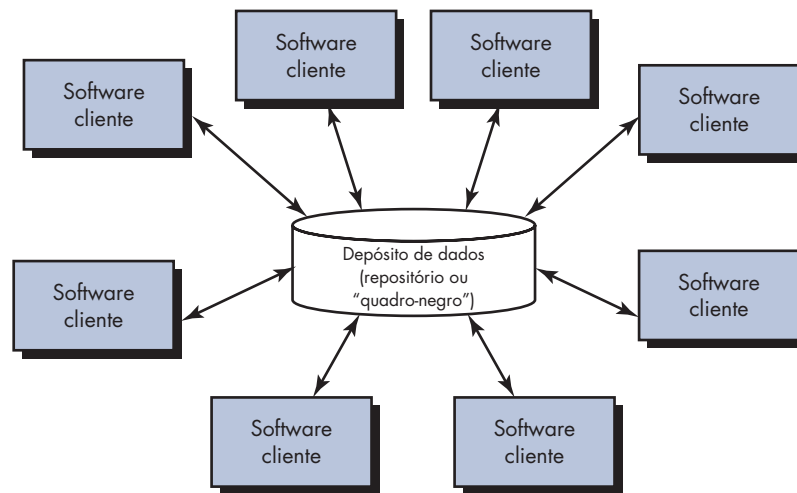
“O uso de padrões e estilos de projeto é universal nas disciplinas de engenharia.”

Mary Shaw e
David Garlan

9.3.1 Uma breve taxonomia dos estilos de arquitetura

Embora milhões de sistemas computacionais tenham sido criados ao longo dos últimos 60 anos, a vasta maioria pode ser classificada em um número relativamente pequeno de estilos de arquitetura:

Arquiteturas centralizadas em dados. Um repositório de dados (por exemplo, um arquivo ou banco de dados) reside no centro dessa arquitetura e é em geral acessado por outros componentes que atualizam, acrescentam, eliminam ou de alguma outra maneira modificam dados contidos no repositório. A Figura 9.1 ilustra um estilo centralizado em dados típico. O software-

FIGURA 9.1**Arquitetura centralizada em dados**

cliente acessa um repositório central. Em alguns casos o repositório de dados é passivo. Ou seja, o software-cliente acessa os dados independentemente de quaisquer alterações nos dados ou das ações de outros softwares-clientes. Uma variação dessa abordagem transforma o repositório em um “quadro-negro” que envia notificações ao software-cliente quando os dados de seu interesse mudam.

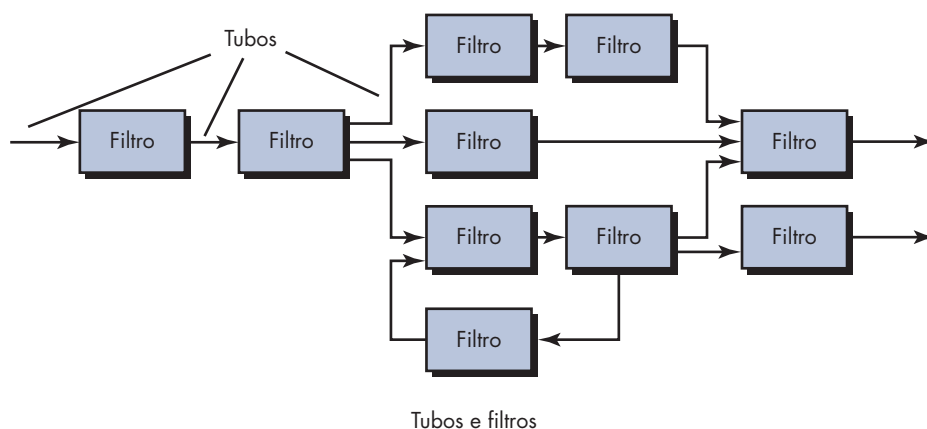
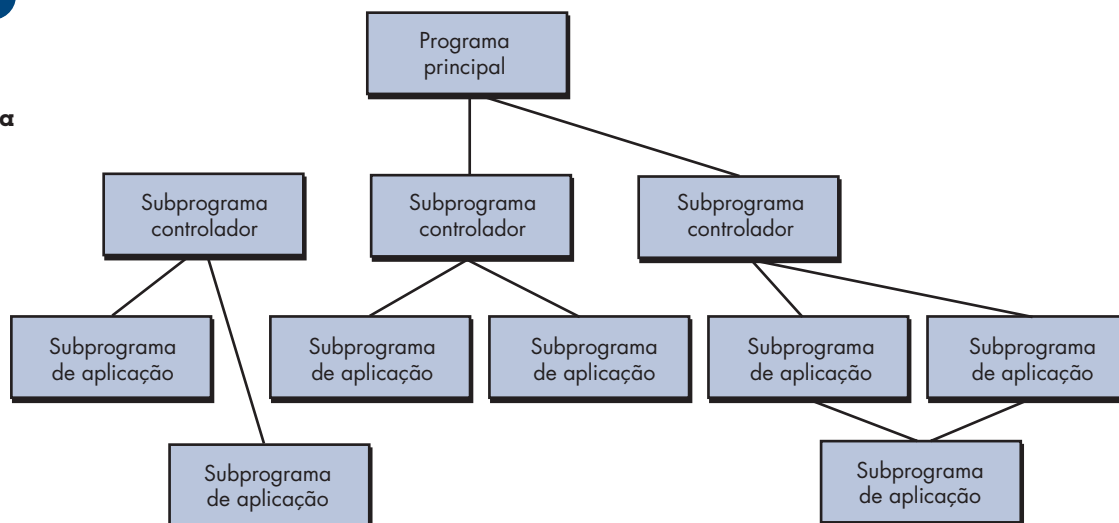
As arquiteturas centralizadas em dados promovem a *integrabilidade* [Bas03]. Isto é, componentes existentes podem ser alterados e novos componentes-clientes acrescentados à arquitetura sem se preocupar com outros clientes (pois os componentes-clientes operam independentemente). Além disso, dados podem ser passados entre os clientes usando o mecanismo de quadro-negro (o componente quadro-negro serve para coordenar a transferência de informações entre os clientes). Os componentes-clientes executam processos de maneira independente.

Arquiteturas de fluxo de dados. Essa arquitetura se aplica quando dados de entrada devem ser transformados por meio de uma série de componentes computacionais ou de manipulação em dados de saída. Um padrão tubos-e-filtro (Figura 9.2) tem um conjunto de componentes, denominado *filtros*, conectados por *tubos* que transmitem dados de um componente para o seguinte. Cada filtro trabalha independentemente dos componentes que se encontram acima e abaixo deles, é projetado para esperar a entrada de dados de determinada forma e produzir saída de dados (para o filtro seguinte) da forma especificada. Entretanto, o filtro não precisa conhecer o funcionamento interno de seus filtros vizinhos.

Se o fluxo de dados ocorrer em uma única linha de transformações, é denominado sequencial por lotes. Essa estrutura aceita um lote de dados e aplica uma série de componentes sequenciais (filtros) para transformá-lo.

Arquiteturas de chamadas e retornos. Esse estilo de arquitetura permite-nos obter uma estrutura de programa relativamente fácil de modificar e aumentar. Existe uma série de subestilos [Bas03] dentro desta categoria:

- *Arquiteturas de programa principal/subprograma.* Essa clássica estrutura de programas decompõe a função em uma hierarquia de controle na qual um programa “principal” invoca uma série de componentes de programa que, por sua vez, podem invocar outros. A Figura 9.3 ilustra uma arquitetura desse tipo.
- *Arquiteturas de chamadas a procedimentos remotos.* Os componentes de uma arquitetura de programas principal /subprogramas são distribuídos ao longo de vários computadores em uma rede.

FIGURA 9.2**Arquitetura de fluxo de dados****FIGURA 9.3****Arquitetura programa principal/subprograma**

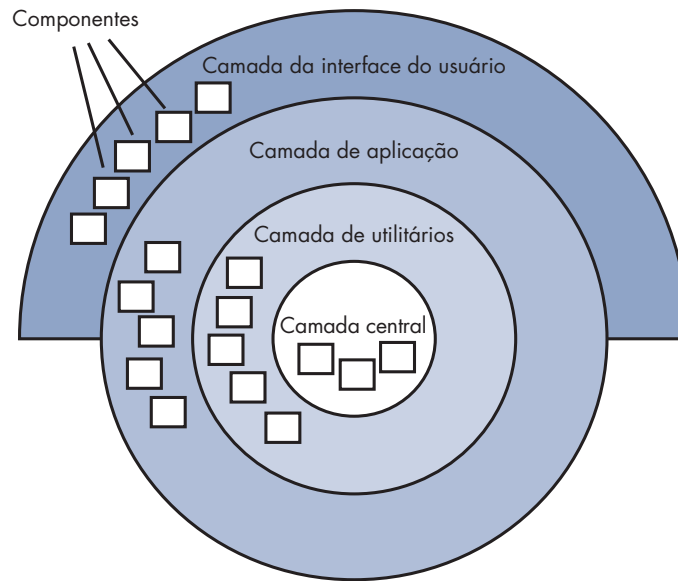
Arquiteturas orientadas a objetos. Os componentes de um sistema encapsulam dados e as operações que devem ser aplicadas para manipular os dados. A comunicação e a coordenação entre componentes são realizadas através da passagem de mensagens.

Arquiteturas em camadas A estrutura básica de uma arquitetura em camadas é ilustrada na Figura 9.4. São definidas várias camadas diferentes, cada uma realizando operações que progressivamente se tornam mais próximas do conjunto de instruções de máquina. Na camada mais externa, os componentes atendem operações de interface do usuário. Na camada mais interna, os componentes realizam a interface com o sistema operacional. As camadas intermediárias fornecem serviços utilitários e funções de software de aplicação.

Esses estilos de arquitetura são apenas um pequeno subconjunto dos disponíveis.² Assim que a engenharia de requisitos revelar as características e restrições do sistema a ser construído, o estilo e/ou combinação de padrões de arquitetura que melhor se encaixa nessas características e restrições pode ser escolhido. Em muitos casos, mais de um padrão poderia ser apropriado, e estilos de arquitetura alternativos podem ser projetados e avaliados. Por exemplo, um estilo em camadas (apropriado para a maioria dos sistemas) pode ser combinado com uma arquitetura centralizada em dados em diversas aplicações de bancos de dados.

2 Veja [Bus07], [Gor06], [Roz05], [Bas03], [Bos00] ou [Hof00] para uma discussão detalhada sobre padrões e estilos de arquitetura.

FIGURA 9.4
Arquitetura em camadas



“Talvez ele esteja no porão. Vou subir e verificar.”

M. C. Escher

9.3.2 Padrões de arquitetura

À medida que o modelo de requisitos é desenvolvido, poderemos perceber que o software deve tratar uma série de problemas mais amplos que envolvem toda a aplicação. Por exemplo, o modelo de requisitos para praticamente qualquer aplicação de comércio eletrônico depara com o seguinte problema: *Como oferecer uma ampla gama de produtos para uma ampla gama de clientes e permitir que esses clientes comprem nossos artigos on-line?*

CASA SEGURA



Escolhendo um estilo de arquitetura

Cena: Sala do Jamie, à medida que a modelagem de projeto começa.

Atores: Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Ed (franzindo a testa): Modelamos a função de segurança usando UML... Você sabe, classes, relações, esse tipo de coisas. Portanto, imagino que a arquitetura³ orientada a objetos seja o caminho a seguirmos.

Jamie: Mas...?

Ed: Mas... Tenho dificuldade em visualizar o que é uma arquitetura orientada a objetos. Entendo a arquitetura de chamadas e retornos, uma espécie de hierarquia de processos convencional, mas orientada a objetos... Eu não sei, ela parece um tanto amorfa.

Jamie (sorrindo): Amorfa, uh?

Ed: Isso mesmo... O que eu quis dizer é que não consigo visualizar uma estrutura real, apenas classes de projeto flutuando no ar.

Jamie: Bem, isso não é verdade. Existem hierarquias de classes... Imagine a hierarquia (agregação) que fizemos para o objeto **Planta** [Figura 8.3]. Uma arquitetura orientada a objetos é uma combinação daquela estrutura e as interconexões — sabe, colaborações — entre as classes. Podemos mostrá-la descrevendo completamente os atributos e operações, a troca de mensagens que ocorre e a estrutura das classes.

Ed: Vou gastar uma hora mapeando uma arquitetura de chamadas e retornos; então voltarei e considerarei uma arquitetura orientada a objetos.

Jamie: Doug não terá nenhum problema com isso. Ele me disse que deveríamos considerar alternativas de arquitetura. Por sinal, não há absolutamente nenhuma razão para que essas duas arquiteturas não possam ser usadas de forma combinada.

Ed: Bom. Eu concordo.

³ Pode ser argumentado que a arquitetura do *CasaSegura* deveria ser considerada em um nível mais elevado do que a arquitetura citada. O *CasaSegura* possui uma série de subsistemas — funcionalidade de monitoramento da casa, o site de monitoramento da empresa e o subsistema que roda no PC do proprietário do imóvel. Nos subsistemas, processos concorrentes (por exemplo, aqueles para monitorar sensores) e tratamento de eventos são frequentes. Algumas decisões em relação à arquitetura nesse nível são tomadas durante a engenharia de produto, porém o projeto da arquitetura na engenharia de software deve considerar muito bem essas questões.

O modelo de requisitos também define um contexto no qual essa questão deve ser respondida. Por exemplo, uma aplicação de comércio eletrônico que vende equipamento de golfe para clientes irá operar em um contexto diferente daquele de uma aplicação de comércio eletrônico que vende equipamentos industriais de preço elevado para empresas de médio e grande porte. Além disso, um conjunto de limitações e restrições poderia afetar a forma que tratamos o problema a ser resolvido.

Os padrões de arquitetura tratam um problema específico de aplicação em um contexto específico e sob um conjunto de limitações e restrições. O padrão propõe uma solução de arquitetura capaz de servir como base para o projeto da arquitetura.

Citamos anteriormente, neste capítulo, que a maioria das aplicações enquadra-se em um domínio ou gênero específico e que um ou mais estilos de arquitetura poderiam ser apropriados para aquele gênero. Por exemplo, o estilo de arquitetura geral para uma aplicação poderia ser de chamadas e retornos ou orientado a objetos. Porém, nesse estilo, encontraremos um conjunto de problemas comuns que poderiam ser mais bem tratados com padrões de arquitetura específicos. Alguns desses problemas e uma discussão mais completa de padrões de arquitetura são apresentados no Capítulo 12.

9.3.3 Organização e refinamento

Pelo fato de o processo de projeto muitas vezes nos dar uma série de alternativas de arquitetura, é importante estabelecer um conjunto de critérios de projeto que podem ser usados para avaliar o projeto da arquitetura obtido. As seguintes questões [Bas03] dão uma visão mais clara sobre um estilo de arquitetura:

 **Como avaliar um estilo de arquitetura obtido?**

Controle. Como o controle é gerenciado na arquitetura? Existe uma hierarquia de controle distinta e, em caso positivo, qual o papel dos componentes nessa hierarquia de controle? Como os componentes transferem controle no sistema? Como o controle é compartilhado entre os componentes? Qual a topologia de controle (ou seja, a forma geométrica que o controle assume)? O controle é sincronizado ou os componentes operam de maneira assíncrona?

Dados. Como os dados são transmitidos entre os componentes? O fluxo de dados é contínuo ou os objetos de dados são passados esporadicamente para o sistema? Qual o modo de transferência de dados (ou seja, os dados são passados de um componente para outro ou os dados estão disponíveis globalmente para ser compartilhados entre os componentes de sistema)? Os componentes de dados (por exemplo, um quadro-negro ou repositório) existem e, em caso positivo, qual o seu papel? Como os componentes funcionais interagem com os componentes de dados? Os componentes de dados são passivos ou ativos (isto é, o componente de dados interage ativamente com outros componentes do sistema)? Como os dados e controle interagem no sistema?

Essas questões dão ao projetista uma avaliação prévia da qualidade do projeto e formam a base para análise mais detalhada da arquitetura.

9.4 PROJETO DA ARQUITETURA

“Um médico pode enterrar seus erros, mas um arquiteto pode apenas aconselhar seu cliente a plantar videiras.”

Frank Lloyd Wright

Quando o projeto da arquitetura se inicia, o software a ser desenvolvido deve ser colocado no contexto — o projeto deveria definir as entidades externas (outros sistemas, dispositivos, pessoas) com as quais o software interage e a natureza da interação. Essas informações em geral podem ser obtidas do modelo de requisitos e todas as demais coletadas durante a engenharia de requisitos. Uma vez que o contexto é modelado e todas as interfaces de software externas tenham sido descritas, podemos identificar um conjunto de arquétipos arquiteturais. *Arquétipo* é uma abstração (similar a uma classe) que representa um elemento do comportamento do sistema. O conjunto de arquétipos fornece uma coleção de abstrações que deve ser modelada arquiteturalmente caso o sistema tenha de ser construído, porém os arquétipos em si não fornecem detalhes de implementação suficientes. Consequentemente, o projetista especifica uma

estrutura do sistema por meio da definição e refinamento de componentes de software que implementam cada arquétipo. Esse processo continua iterativamente até que uma estrutura de arquitetura completa tenha sido obtida. Nas seções seguintes examinaremos cada uma dessas tarefas de projeto da arquitetura com um pouco mais de detalhes.

9.4.1 Representação do sistema no contexto

PONTO-CHAVE

O contexto de arquitetura representa como o software interage com entidades externas a seus limites.

Como os sistemas operam entre si?

No nível de projeto da arquitetura, um arquiteto de software usa um *diagrama de contexto arquitetural* (*architectural context diagram, ACD*) para modelar a maneira pela qual o software interage com entidades externas a seus limites. A estrutura genérica do diagrama de contexto arquitetural é ilustrada na Figura 9.5.

Referindo-se à figura, os sistemas que interoperam com o *sistema-alvo* (o sistema para o qual um projeto da arquitetura deve ser desenvolvido) são representados como

- *Sistemas superiores* — aqueles sistemas que usam o sistema-alvo como parte de algum esquema de processamento de mais alto nível.
- *Sistemas subordinados* — aqueles sistemas que são utilizados pelo sistema-alvo e fornecem dados ou processamento necessários para completar a funcionalidade do sistema-alvo.
- *Sistemas de mesmo nível (pares)* — aqueles sistemas que interagem em uma base par-a-par (ou seja, as informações são produzidas ou consumidas pelos pares e sistema-alvo).
- *Atores* — entidades (pessoas, dispositivos) que interagem com o sistema-alvo através da produção ou consumo de informações necessárias para o processamento.

Cada uma dessas entidades externas se comunica com o sistema-alvo por meio de uma interface (os pequenos retângulos sombreados). Para ilustrarmos o uso do ACD, consideremos a função de segurança residencial do produto *CasaSegura*. O controlador geral do produto *CasaSegura* e o sistema baseado na Internet são ambos superiores em relação à função de segurança e são mostrados acima da função na Figura 9.6. A função de vigilância é um *sistema de mesmo nível* e utiliza (é utilizado por) a função de segurança residencial em versões posteriores do produto. O proprietário do imóvel e os painéis de controle são os atores que agem tanto como produtores quanto consumidores de informações usadas/produzidas pelo software de segurança. Por fim, são utilizados sensores pelo software de segurança e mostrados como subordinados a ele.

FIGURA 9.5

Diagrama de contexto arquitetural

Fonte: Adaptado de (Bos00)

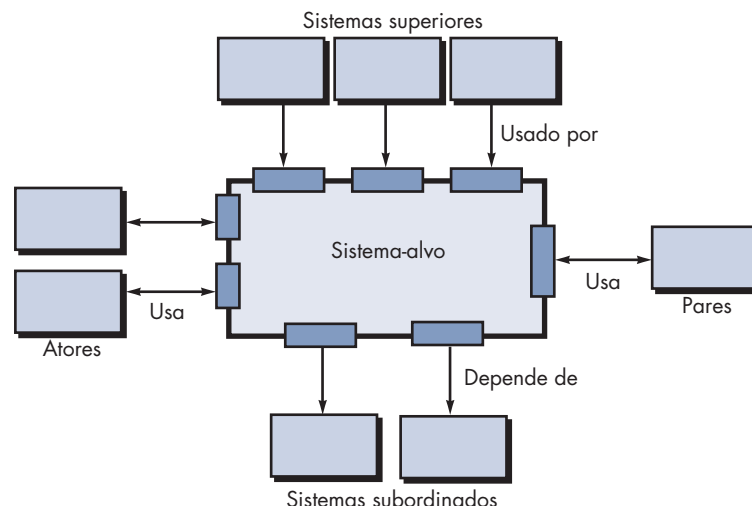
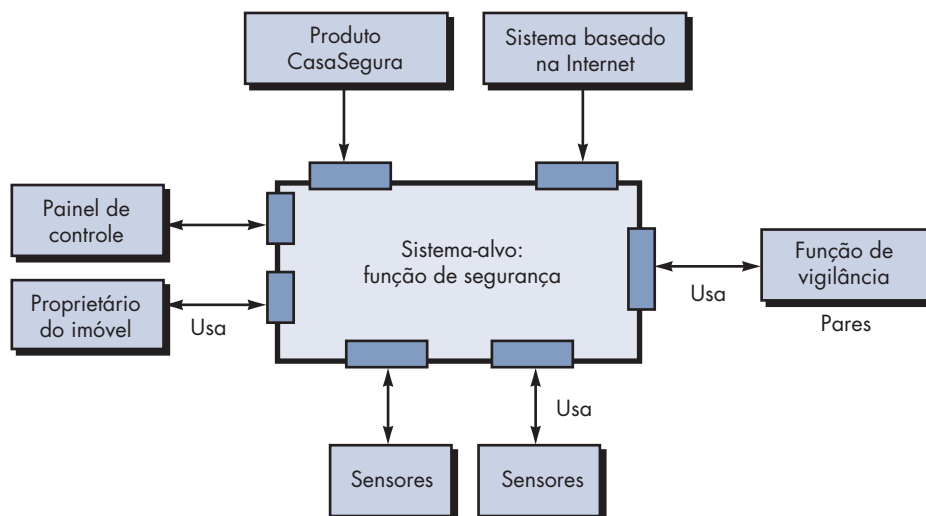


FIGURA 9.6

Diagrama de contexto arquitetural para a função de segurança do CasaSegura



Como parte do projeto da arquitetura, os detalhes de cada interface da Figura 9.6 teriam de ser especificados. Todos os dados que fluem para dentro e para fora do sistema-alvo têm de ser identificados neste estágio.

9.4.2 Definição de arquétipos

Arquétipo é uma classe ou padrão que representa uma abstração central crítica para o projeto de uma arquitetura para o sistema-alvo. Em geral, é necessário um conjunto relativamente pequeno de arquétipos para projetar até mesmo sistemas relativamente complexos. A arquitetura do sistema-alvo é composta desses arquétipos, que representam elementos estáveis da arquitetura, porém podem ser instanciados de várias maneiras tomando como base o comportamento do sistema.

Em muitos casos, os arquétipos podem ser derivados examinando-se as classes de análise definidas como parte do modelo de requisitos. Prosseguindo com a discussão da função de segurança domiciliar do *CasaSegura*, poderíamos definir os seguintes arquétipos:

- **Nó.** Representa um conjunto coeso de elementos de entrada e saída da função de segurança domiciliar. Por exemplo, um nó poderia ser composto por (1) vários sensores e (2) uma série de indicadores (de saída) de alarme.
- **Detector.** Abstração que engloba todos os equipamentos de sensoriamento que alimentam o sistema-alvo com informações.
- **Indicador.** Abstração que representa todos os mecanismos (por exemplo, sirene de alarme, luzes intermitentes, campainha) para indicar a ocorrência de condição de alarme.
- **Controlador.** Abstração que representa o mecanismo que permite armar ou desarmar um nó. Se os controladores residirem em uma rede, eles têm a habilidade de se comunicarem entre si.

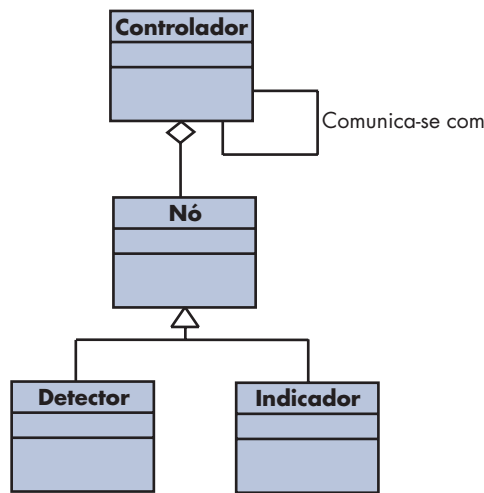
Cada um dos arquétipos é representado usando-se a notação UML conforme indicado na Figura 9.7. Recorde-se que os arquétipos formam a base para a arquitetura, mas são as abstrações que devem ser refinadas à medida que o projeto da arquitetura prossegue. Por exemplo, **Detector** poderia ser refinado em uma hierarquia de classes de sensores.

PONTO-CHAVE

Arquétipos são os blocos básicos abstratos de um projeto da arquitetura.

FIGURA 9.7

Relacionamentos UML para os arquétipos da função de segurança do CasaSegura
 Fonte: Adaptado de (Bos00)



9.4.3 Refinamento da arquitetura em componentes

À medida que a arquitetura de software é refinada em componentes, a estrutura do sistema começa a emergir. Mas como os componentes são escolhidos? Para podermos responder a essa pergunta, começamos pelas classes descritas como parte do modelo de requisitos.⁴ As classes de análise representam entidades no domínio de aplicação que deve ser tratado na arquitetura de software. Portanto, o domínio de aplicação é uma fonte para a derivação e o refinamento de componentes. Outra fonte é o domínio da infraestrutura. A arquitetura deve acomodar muitos componentes de infraestrutura que possibilitem componentes de aplicação, mas que não tenham nenhuma ligação de atividade com o domínio de aplicação. Por exemplo, componentes de gerenciamento de memória, componentes de comunicação, componentes de bancos de dados e componentes de gerenciamento de tarefas em geral são integrados à arquitetura do software.

As interfaces representadas no diagrama de contexto arquitetural (Seção 9.4.1) implica um ou mais componentes especializados que processam os dados que fluem pela interface. Em alguns casos (por exemplo, uma interface gráfica do usuário), tem de ser projetada uma arquitetura de subsistemas completa com vários componentes.

Continuando com o exemplo da função de segurança domiciliar do *CasaSegura*, poderíamos definir o conjunto de componentes de alto nível que atende à seguinte funcionalidade:

- *Gerenciamento da comunicação externa* — coordena a comunicação da função de segurança com entidades externas como sistemas baseados na Internet e notificação externa de alarme.
- *Processamento de painel de controle* — gerencia toda a funcionalidade do painel de controle.
- *Gerenciamento de detectores* — coordena o acesso a todos os detectores conectados ao sistema.
- *Processamento de alarme* — verifica e atua sobre todas as condições de alarme.

Cada um dos componentes de alto nível teria de ser elaborado de forma iterativa e posicionado na arquitetura global do *CasaSegura*. As classes de projeto (com atributos e operações

“A estrutura de um sistema de software fornece o meio no qual o código nasce, amadurece e morre. Um hábitat bem projetado possibilita a evolução bem-sucedida de todos os componentes necessários em um sistema de software.”

R. Pattis

⁴ Se optar-se por uma abordagem convencional (não orientada a objetos), os componentes são obtidos do modelo de fluxo de dados. Discutiremos brevemente esse método na Seção 9.6.

apropriados) seriam definidas para cada um deles. Entretanto, é importante notar que os detalhes de projeto de todos os atributos e operações não seriam especificados até se atingir o projeto de componentes (Capítulo 10).

A estrutura global de arquitetura (representada na forma de um diagrama de componentes UML) é ilustrada na Figura 9.8. As transações são capturadas por *gerenciamento de comunicação externa* à medida que se deslocam de componentes que processam a interface gráfica do usuário do *CasaSegura* e a interface com a Internet. Tais informações são gerenciadas por um componente executivo do *CasaSegura* que seleciona a função do produto apropriada (no caso, segurança). O componente *processamento do painel de controle* interage com o proprietário do imóvel para armar/desarmar a função de segurança. O componente *gerenciamento de detectores* faz uma sondagem nos sensores para detectar condição de alarme, e o componente de *processamento de alarme* gera uma saída quando o alarme é detectado.

9.4.4 Descrição das instâncias

O projeto da arquitetura modelado até este ponto ainda é relativamente de alto nível. O contexto do sistema foi representado, arquétipos que indicam importantes abstrações contidas no domínio do problema foram definidos, a estrutura global do sistema está evidente e os principais componentes de software foram identificados. Entretanto, um maior refinamento (recorde-se que todo projeto é iterativo) ainda é necessário.

Para tanto, é desenvolvida uma instância real da arquitetura. Queremos dizer com isso que a arquitetura é aplicada a um problema específico com o intuito de demonstrar que a estrutura e os componentes são apropriados.

A Figura 9.9 ilustra uma instância da arquitetura do *CasaSegura* para o sistema de segurança. Os componentes da Figura 9.8 são elaborados para indicar mais detalhes. Por exemplo, o componente *gerenciamento de detectores* interage com o *componente de infraestrutura agendador* que implementa a sondagem (*pooling*) de cada objeto *sensor* usado pelo sistema de segurança. Elaboração similar é feita para cada um dos componentes representados na Figura 9.8.

FIGURA 9.8

Estrutura de arquitetura global para o sistema *CasaSegura* com os componentes de alto nível

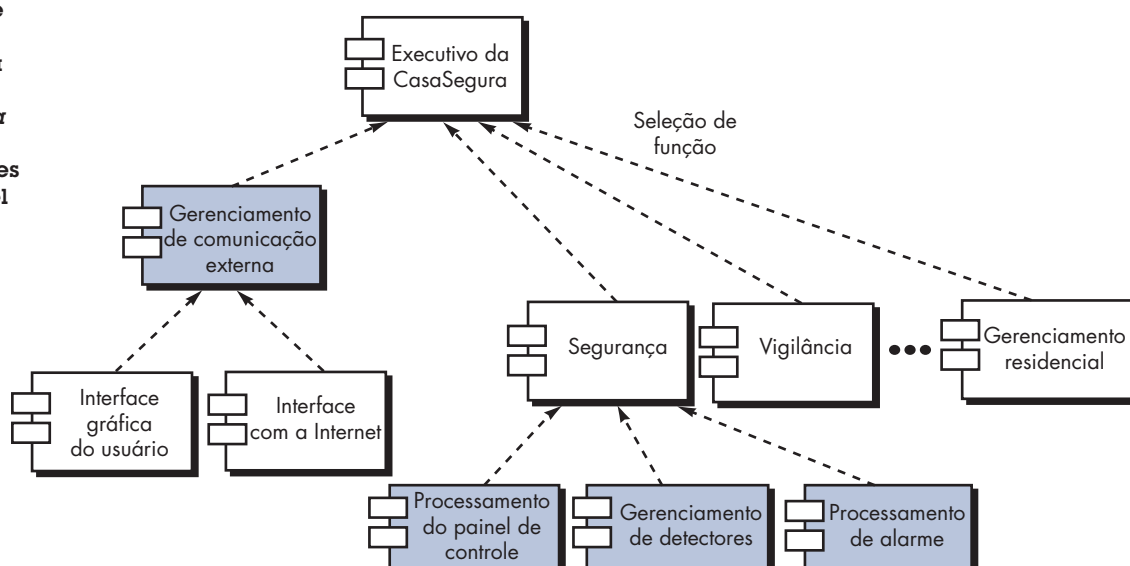
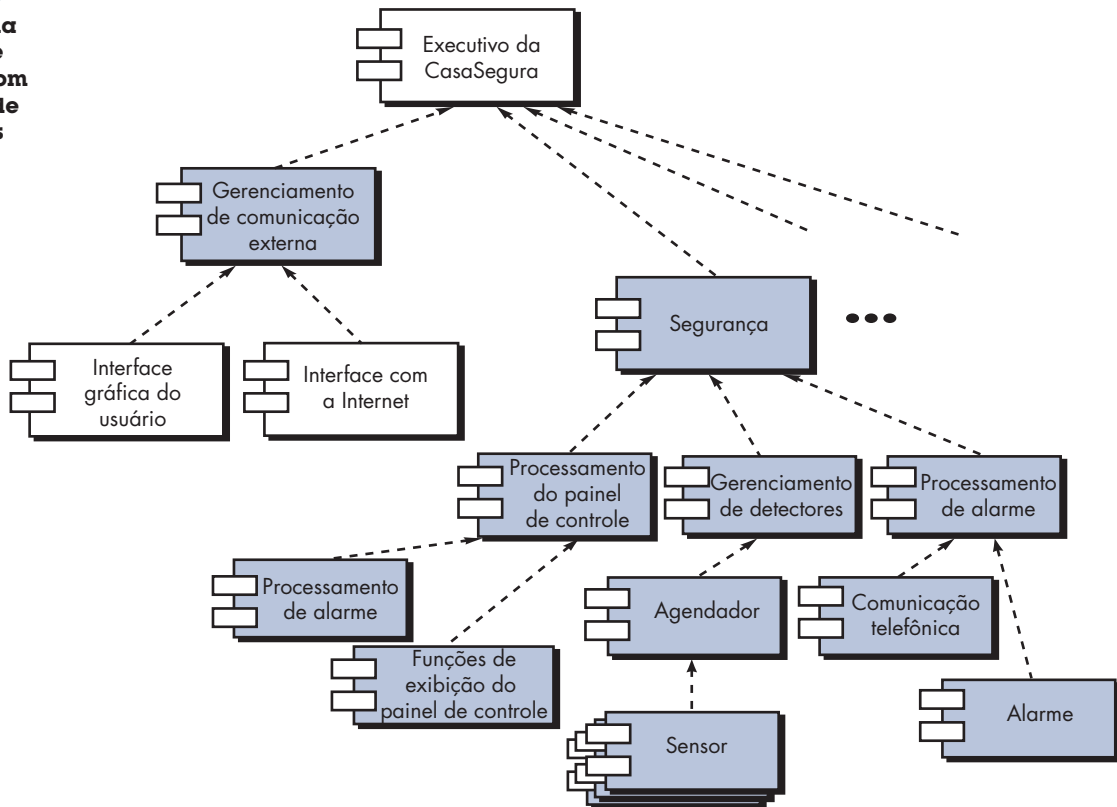


FIGURA 9.9

Uma instância da função de segurança com elaboração de componentes



FERRAMENTAS DO SOFTWARE



Projeto da arquitetura

Objetivo: As ferramentas de projeto da arquitetura modelam a estrutura global do software, representando a interface com componentes, as dependências e relações, bem como as interações.

Mecânica: A mecânica das ferramentas varia. Na maioria dos casos, a capacidade de projeto da arquitetura faz parte da funcionalidade fornecida por ferramentas automatizadas para análise e modelagem de projeto.

Ferramentas representativas:⁵

Adalon, desenvolvida pela Synthis Corp. (www.synthis.com), é uma ferramenta de projeto especializada para

o projeto e a construção de arquiteturas específicas de componentes baseados na Web.

ObjectiF, desenvolvida pela microTOOL GmbH (www.microtool.de/objectiF/en/), é uma ferramenta de projeto baseada em UML que conduz a arquiteturas (por exemplo, Coldfusion, J2EE, Fusebox) suscetíveis à engenharia de software baseada em componentes (Capítulo 29).

Rational Rose, desenvolvida pela Rational (www-306.ibm.com/software/rational/), é uma ferramenta de projeto baseada em UML que oferece suporte a todos os aspectos do projeto da arquitetura.

9.5 AVALIAÇÃO DE PROJETOS DE ARQUITETURAS ALTERNATIVOS

Em seu livro sobre avaliação de arquiteturas de software, Clements e seus colegas [Cle03] afirmam:

Grosso modo, arquitetura é uma aposta, uma aposta no sucesso de um sistema. Não seria ótimo saber antecipadamente se apostamos em um vencedor, em vez de ter de esperar até que o sistema esteja

⁵ As ferramentas aqui apresentadas não significam um aval, mas sim, uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

quase completo antes de saber se ele irá atender ou não seus requisitos? Caso esteja adquirindo um sistema ou pagando por seu desenvolvimento, você não gostaria de ter alguma certeza de que foi iniciado de maneira correta? Caso seja o próprio arquiteto de sistemas, você não gostaria de ter um bom meio de validar suas intuições e experiência, para poder dormir à noite sabendo que a confiança colocada em seu projeto é bem fundamentada?

De fato, respostas a essas questões teriam valor. O projeto resulta em uma série de alternativas de arquitetura, cada uma das quais avaliadas para determinar qual delas é a mais apropriada para o problema a ser resolvido. Nas seções a seguir, apresentaremos duas abordagens diferentes para a avaliação de projetos de arquitetura alternativos. A primeira usa um método iterativo para avaliar prós e contras de projeto. A segunda abordagem aplica uma técnica pseudoquantitativa para avaliar a qualidade de um projeto.

9.5.1 Um método de análise dos prós e contras de uma arquitetura

O SEI (Software Engineering Institute) desenvolveu um *método de análise dos prós e contras de uma arquitetura* (ATAM, *architecture trade-off analysis method*, ATAM) [Kaz98] que estabelece um processo de avaliação iterativa de arquiteturas de software. As atividades de análise de projeto a seguir são realizadas iterativamente:

1. *Coletar cenários.* É desenvolvido um conjunto de casos de uso (Capítulos 5 e 6) para representar o sistema sob o ponto de vista do usuário.
2. *Elicitar requisitos, restrições e descrição do ambiente.* Essas informações são determinadas como parte da engenharia de requisitos e usadas para haver certeza de que todas as necessidades dos interessados foram atendidas.
3. *Descrever os estilos/padrões de arquitetura escolhidos para lidar com os cenários e requisitos.* O(s) estilo(s) de arquitetura deveria(m) ser descrito(s) usando uma das seguintes visões de arquitetura:
 - *Visão de módulos* para a análise de atribuições de trabalho com componentes e o grau que foi atingido pelo encapsulamento de informações.
 - *Visão de processos* para a análise do desempenho do sistema.
 - *Visão de fluxo de dados* para análise do grau que a arquitetura atende às necessidades funcionais.
4. *Avaliar atributos de qualidade considerando cada atributo isoladamente.* O número de atributos de qualidade escolhidos para análise é em função do tempo disponível para revisão e o grau em que os atributos de qualidade são relevantes para o sistema em questão. Entre os atributos de qualidade para avaliação de projetos de arquitetura temos confiabilidade, desempenho, segurança, facilidade de manutenção, flexibilidade, facilidade de teste, portabilidade, reusabilidade e interoperabilidade.
5. *Identificar a sensibilidade dos atributos de qualidade em relação a vários atributos de arquitetura para um estilo de arquitetura específico.* Isso pode ser obtido fazendo-se pequenas alterações na arquitetura e na determinação de quão sensível um atributo de qualidade, digamos o desempenho, é em relação a uma mudança. Quaisquer atributos afetados significativamente por uma variação na arquitetura são denominados *pontos de sensibilidade*.
6. *Criticar arquiteturas candidatas (desenvolvidas na etapa 3) usando a análise de sensibilidade conduzida na etapa 5.* O SEI descreve esse método da seguinte maneira [Kaz98]:

Uma vez determinados os pontos de sensibilidade da arquitetura, encontrar os prós e os contras é simplesmente a identificação de elementos de arquitetura para os quais vários atributos são sensíveis. Por exemplo, o desempenho de uma arquitetura cliente/servidor poderia ser altamente sensível ao número de servidores (o desempenho aumenta, em algum intervalo, aumentando-se o número de servidores)... O número de servidores é então um ponto de avaliação com relação a essa arquitetura.

As seis etapas representam a primeira iteração ATAM. Baseado nos resultados das etapas 5 e 6, algumas alternativas de arquitetura poderiam ser eliminadas, uma ou mais arquiteturas

WebRef

Informações detalhadas sobre o ATAM podem ser obtidas em:

www.sei.cmu.edu/activities/architecture/ata_method.html.

CASA SEGURA



Avaliação de arquiteturas

Cena: Sala de Doug Miller à medida que a modelagem do projeto da arquitetura prossegue.

Atores: Vinod, Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura* e Doug Miller, gerente do grupo de engenharia de software.

Conversa:

Doug: Sei que vocês estão obtendo uma série de arquiteturas diferentes para o produto *CasaSegura*, e isso é bom. Acredito que a minha pergunta seja “como iremos escolher a melhor delas”?

Ed: Estou trabalhando em um estilo de chamadas e retornos e depois eu ou o Jamie vamos derivar uma arquitetura orientada a objetos.

Doug: Certo, e como fazemos essa escolha?

Jamie: Fiz um curso de projeto no meu último ano de faculdade e me recordo que há uma série de maneiras para fazer isso.

Vinod: Realmente, porém elas são um tanto acadêmicas. Veja, acredito que possamos fazer nossa avaliação e escolher a arquitetura correta empregando casos de uso e cenários.

Doug: Isso não é a mesma coisa?

Vinod: Não quando se trata de avaliação de arquiteturas. Já temos um conjunto completo de casos de uso. Portanto, aplica-

mos cada um deles a ambas as arquiteturas e verificamos como o sistema reage, como os componentes e conectores funcionam no contexto dos casos de uso.

Ed: É uma boa ideia. Garante que não nos esqueçamos de nada.

Vinod: É verdade, mas eles também nos informam se o projeto da arquitetura é complicado ou não, se o sistema tem ou não de se desdobrar para conseguir realizar sua tarefa.

Jamie: Cenários não é justamente um outro nome para casos de uso.

Vinod: Não, neste caso um cenário implica algo diferente.

Doug: Você está se referindo a um cenário de qualidade ou um cenário de mudanças, certo?

Vinod: Isso mesmo. O que fazemos é procurar novamente os interessados e perguntar a eles como provavelmente o *CasaSegura* irá se modificar ao longo dos próximos, digamos, três anos. Sabe, novas versões, recursos, esse tipo de coisa. Construímos um conjunto de cenários de mudanças. Também desenvolvemos um conjunto de cenários de qualidade que definem os atributos que gostaríamos de ver na arquitetura do software.

Jamie: E os aplicamos às alternativas.

Vinod: Exatamente. O estilo que se comportar melhor nos casos de uso e cenários será o escolhido.

remanescentes poderiam ser modificadas e representadas de forma mais detalhada e então as etapas do ATAM seriam reaplicadas.⁶

9.5.2 Complexidade da arquitetura

Uma técnica útil para avaliar a complexidade geral de uma arquitetura proposta é considerar as dependências entre os componentes contidos na arquitetura. Tais dependências são orientadas por fluxo de controle/informações no sistema. Zhao [Zha98] sugere três tipos de dependências:

As dependências de compartilhamento representam relações de dependência entre consumidores que usam o mesmo recurso ou produtores que produzem para os mesmos consumidores. Por exemplo, para dois componentes **u** e **v**, se **u** e **v** fazem referência aos mesmos dados globais, então existe uma relação de dependência compartilhada entre **u** e **v**.

As dependências de fluxo representam relações de dependência entre produtores e consumidores de recursos. Por exemplo, para dois componentes **u** e **v**, se **u** deve completar antes que o controle passe para **v** (pré-requisito), ou se **u** se comunica com **v** através de parâmetros, então existe uma relação de dependência de fluxos entre **u** e **v**.

As dependências de restrição representam restrições no fluxo relativo de controle entre um conjunto de atividades. Por exemplo, para dois componentes **u** e **v**, **u** e **v** não podem ser executados ao mesmo tempo (exclusão mútua), então existe aí uma relação de dependência de restrição entre **u** e **v**.

As dependências de compartilhamento e de fluxo citadas por Zhao são similares ao conceito de acoplamento discutido no Capítulo 8. Acoplamento é um importante conceito de projeto apli-

⁶ O Método de Análise de Arquitetura via Software (Software Architecture Analysis Method, SAAM) é uma alternativa ao ATAM e vale a pena ser examinado por leitores interessados em análise de arquiteturas. Um artigo sobre o SAAM pode ser baixado de www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html.

cável nos níveis de arquitetura e de componente. Métricas simples para avaliar o acoplamento são discutidas no Capítulo 23.

9.5.3 Linguagens de descrição da arquitetura

O arquiteto de uma casa possui um conjunto de notações e ferramentas padronizadas que permitem que o projeto seja representado de uma forma inequívoca e compreensível. Embora o arquiteto de software possa desenhar em notação UML, outras formas esquemáticas e algumas ferramentas relacionadas, há a necessidade de uma abordagem mais formal para a especificação de um projeto da arquitetura.

A *linguagem de descrição da arquitetura* (*architectural description language, ADL*) fornece uma sintaxe e semântica para descrever uma arquitetura de software. Hofmann e seus colegas [Hof01] sugerem que uma ADL deveria fornecer ao projetista a habilidade de decompor componentes da arquitetura, compor componentes individuais em blocos arquiteturais maiores e representar interfaces (mecanismos de conexão) entre componentes. Uma vez estabelecidas técnicas descritivas e baseadas em linguagens para o projeto da arquitetura, é mais provável que métodos de avaliação de arquitetura mais efetivos serão estabelecidos à medida que o projeto evolui.



Linguagens de descrição da arquitetura

O resumo a seguir de uma série de importantes ADLs foi preparado por Rickard Land [Lan02] e é reimpresso com a permissão do autor. Deve-se salientar que as primeiras cinco ADLs listadas foram desenvolvidas para fins de pesquisa e não são produtos comerciais.

Rapide (<http://poset.stanford.edu/rapide/>) baseia-se no conceito de conjuntos ordenados parciais e, portanto, introduz construtos de programação bastante novos (po- rém aparentemente poderosos).

UniCon (www.cs.cmu.edu/~UniCon) é “uma linguagem de descrição da arquitetura destinada a auxiliar os projetistas a definir arquiteturas de software em termos de abstrações que achem úteis”.

Aesop (www.cs.cmu.edu/~able/aesop/) trata o problema de reutilização de estilos. Com a Aesop, é pos-

sível definir estilos e usá-los ao construir um sistema real.

Wright (www.cs.cmu.edu/~able/wright/) é uma linguagem formal que inclui os seguintes elementos: *componentes* com *portas*, *conectores* com *papéis*, e *cola* para ligar papéis às portas. Os estilos de arquitetura podem ser formalizados na linguagem através de predicados permitindo, portanto, verificações estáticas para determinar a consistência e a completude de uma arquitetura.

Acme (www.cs.cmu.edu/~acme/) pode ser vista como uma ADL de segunda geração, já que seu intuito é identificar um tipo de mínimo denominador comum para ADLs.

UML (www.uml.org/) inclui muitos dos artefatos necessários para descrições de arquiteturas — processos, nós, visões etc. Para descrições informais, a UML é adequada apenas pelo fato de ser um padrão largamente compreendido. Falta-lhe, entretanto, o poder necessário para uma descrição adequada de uma arquitetura.

FERRAMENTAS DO SOFTWARE

9.6 MAPEAMENTO DE ARQUITETURA UTILIZANDO FLUXO DE DADOS

Os estilos de arquitetura discutidos na Seção 9.3.1 representam arquiteturas radicalmente diferentes. Portanto, não é de surpreender se um mapeamento abrangente que realize a transição do modelo de requisitos para uma série de estilos de arquitetura não exista. De fato, não existe nenhum mapeamento prático para alguns estilos de arquitetura, e o projetista deve abordar a tradução de requisitos em projeto para esses estilos usando as técnicas discutidas na Seção 9.4.

Para ilustrarmos uma abordagem para mapeamento de arquitetura, consideremos a arquitetura de chamadas e retornos — uma estrutura especial extremamente comum para muitos tipos de sistemas. A arquitetura de chamadas e retornos pode residir no interior de outras arquiteturas mais sofisticadas discutidas anteriormente neste capítulo. Por exemplo, a arquitetura de um ou mais componentes de uma arquitetura cliente/servidor poderia ser de chamadas e retornos.

Uma técnica de mapeamento, chamada *projeto estruturado* [You79], é muitas vezes caracterizada como um fluxo de dados – método de projeto orientado, pois fornece uma transição

conveniente de um diagrama de fluxo de dados (Capítulo 7) para uma arquitetura de software.⁷ A transição de fluxo de informações (representado na forma de um DFD, diagrama de fluxo de dados) para estrutura de programas é obtida como parte de um processo de seis etapas: (1) é estabelecido o tipo de fluxo de informações, (2) são indicadas as fronteiras do fluxo, (3) o DFD é mapeado em uma estrutura de programas, (4) é definida uma hierarquia de controle, (5) a estrutura resultante é refinada usando-se heurística e medidas de projeto e, por fim, (6) a descrição da arquitetura é refinada e elaborada.

Como um rápido exemplo de mapeamento de fluxo de dados, apresentamos um mapeamento “de transformação” para uma pequena parte da função de segurança do *CasaSegura*.⁸ Para realizar o mapeamento, o tipo de fluxo de informações deve ser determinado. Um tipo de fluxo de informações é denominado *fluxo de transformação* e que apresenta uma característica linear. Os dados fluem para dentro do sistema ao longo de uma *trajetória de fluxo de entrada* na qual são transformados de uma representação do mundo exterior em uma forma internalizada. Assim que tiverem sido internalizados, são processados em um *centro de transformação*. Por fim, fluem para fora do sistema ao longo de uma *trajetória de fluxo de saída* que transforma os dados para a forma do mundo exterior.⁹

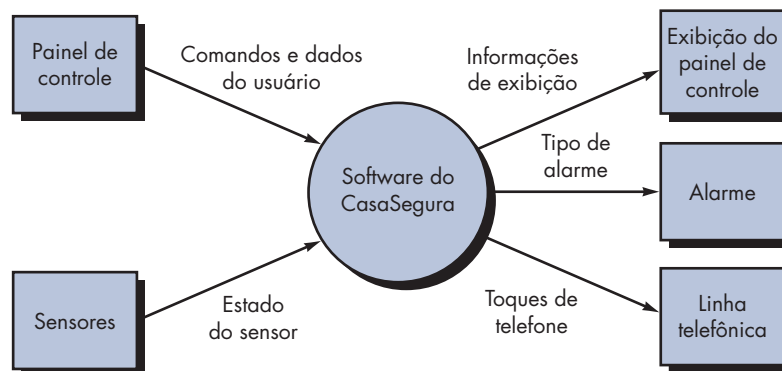
9.6.1 Mapeamento de transformação

Mapeamento de transformação é um conjunto de etapas de projeto que permite que um DFD com características de fluxo de transformação seja mapeado em um estilo de arquitetura específico. Para ilustrarmos essa abordagem, consideraremos mais uma vez a função de segurança do *CasaSegura*.¹⁰ Um elemento do modelo de análise é um conjunto de diagramas de fluxo de dados que descreve o fluxo de informações dentro da função de segurança. Para mapearmos os diagramas de fluxo de dados em uma arquitetura de software, iniciariamos com as seguintes etapas de projeto:

Passo 1. Revisar o modelo fundamental do sistema. O modelo fundamental do sistema ou diagrama de contexto representa a função de segurança como uma única transformação, representando os produtores e consumidores externos de dados que fluem para dentro e para fora da função. A Figura 9.10 representa um modelo de contexto de nível 0, e a Figura 9.11 mostra o fluxo de dados refinado para a função de segurança.

FIGURA 9.10

DFD de nível de contexto para a função de segurança do CasaSegura



⁷ Deve-se notar que outros elementos do modelo de requisitos também são usados durante o método de mapeamento.

⁸ Uma discussão mais detalhada de projeto estruturado é apresentada no site que acompanha este livro.

⁹ Outro importante tipo de fluxo de informações, o *fluxo de transações*, não é considerado nesse exemplo, mas é tratado em um exemplo de projeto estruturado apresentado no site que acompanha este livro.

¹⁰ Consideraremos apenas a parte da função de segurança do *CasaSegura* que utiliza o painel de controle. Outras características discutidas ao longo deste livro não são consideradas aqui.



Se o DFD for refinado ainda mais nesta ocasião, tente ao máximo obter bolhas que exibam alta coesão.

Passo 2. Revisar e refinar diagramas de fluxo de dados para o software. As informações obtidas do modelo de requisitos são refinadas para produzir detalhes significativos. Por exemplo, o DFD de nível 2 para *monitorar sensores* (Figura 9.12) é examinado, e um diagrama de fluxo de dados nível 3 é obtido, conforme mostrado na Figura 9.13. No nível 3, cada transformação no diagrama de fluxo de dados exibe uma coesão relativamente elevada (Capítulo 8). Isto é, o processo correspondente a uma transformação realiza uma única função distinta que pode ser implementada na forma de um componente no software *CasaSegura*. Consequentemente, o DFD da Figura 9.13 contém detalhes suficientes para uma “primeira aproximação” do projeto de arquitetura para o subsistema *monitorar sensores*, e prosseguirmos sem refinamentos adicionais.

FIGURA 9.11

DFD de nível 1 para a função de segurança do CasaSegura

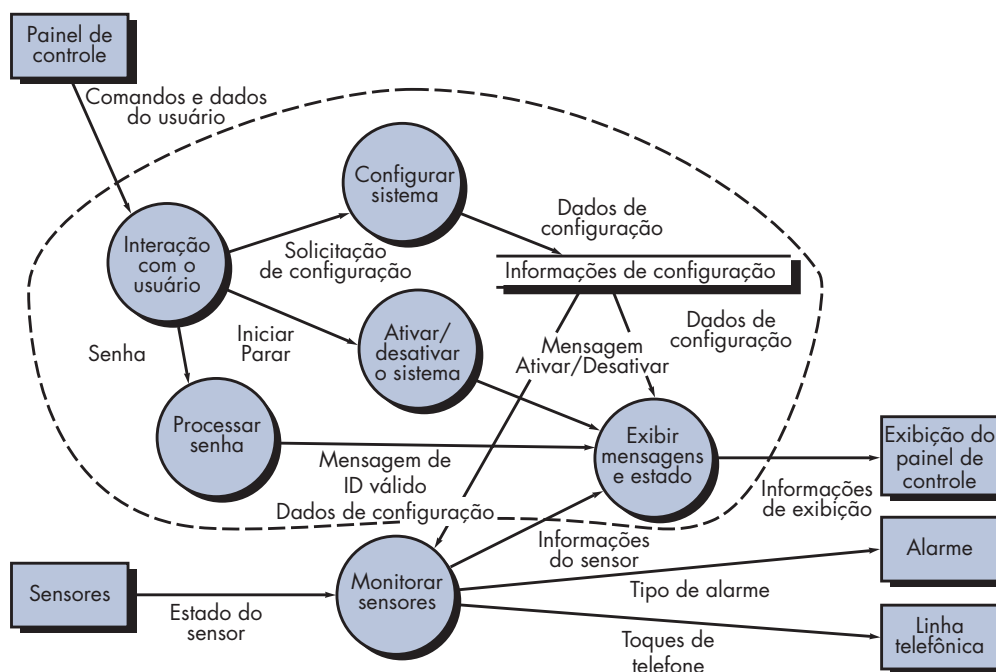
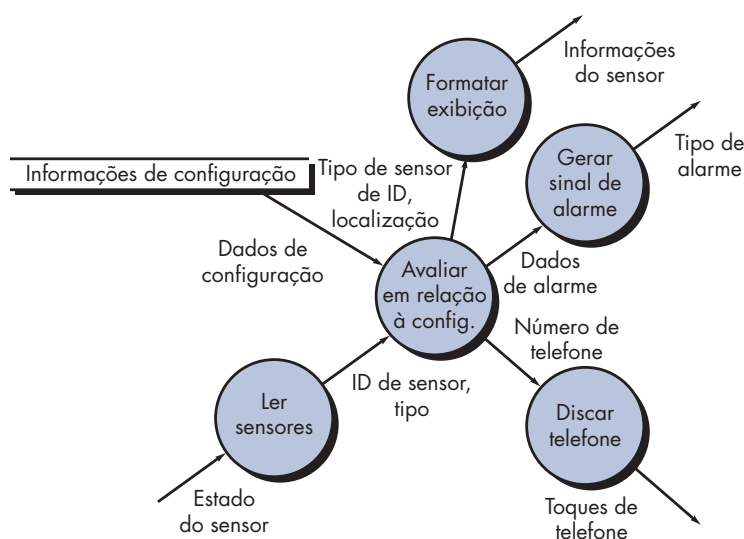


FIGURA 9.12

DFD de nível 2 que refina a transformação de monitorar sensores



PONTO-CHAVE

Muitas vezes encontraremos ambos os tipos de fluxo de dados no interior do mesmo modelo orientado a fluxos. Os fluxos são subdivididos e a estrutura de programas é obtida usando-se o mapeamento apropriado.



Varie a posição das fronteiras dos fluxos em uma tentativa de explorar estruturas alternativas de programas. Isso toma muito pouco tempo e dá uma visão importante do processo.

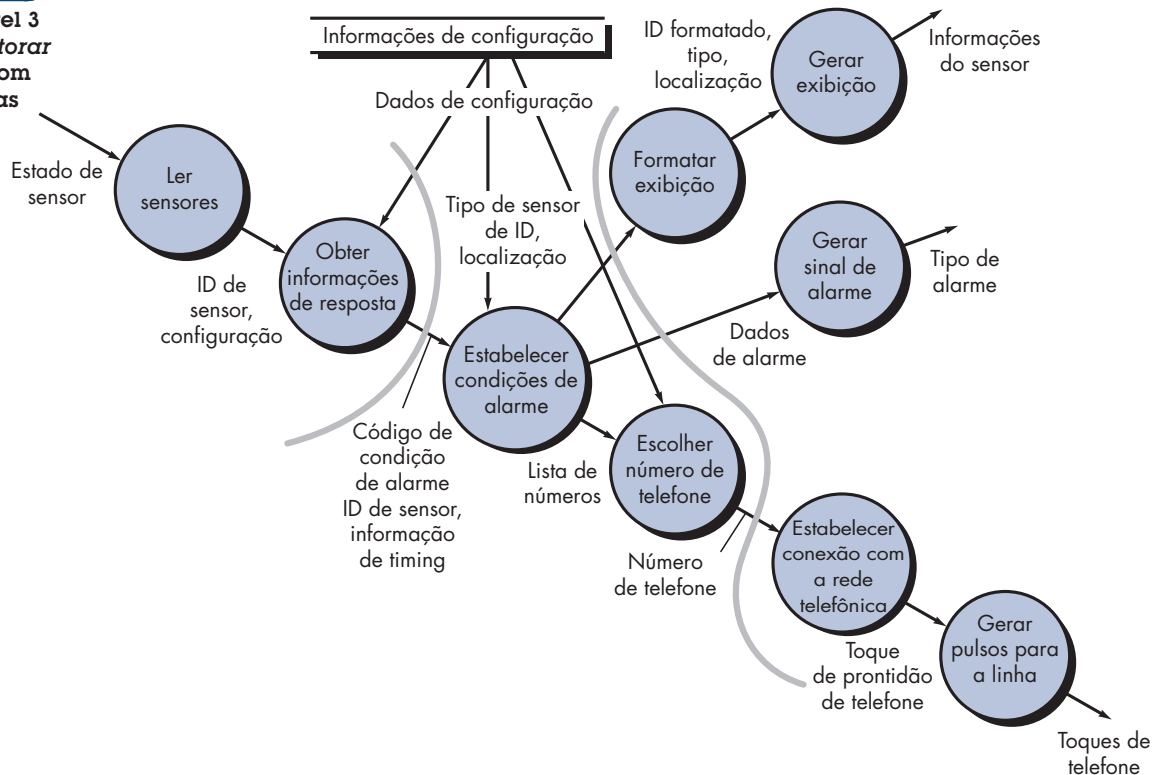
Passo 3. Determinar se o DFD possui características de fluxo de transformação ou transação¹¹. Avaliando-se o DFD (Figura 9.13), podemos ver dados entrando no software ao longo de uma trajetória de entrada e saindo ao longo de três trajetórias de saída. Consequentemente, uma característica de transformação geral será considerada para o fluxo de informações.

Passo 4. Isolar o centro de transformação especificando as fronteiras do fluxo de entrada e saída. Fluxos de dados de entrada ao longo de uma trajetória em que as informações são convertidas da forma externa para a forma interna; o fluxo que sai converte dados internalizados para a forma externa. As fronteiras do fluxo de entrada e saída são de livre interpretação. Ou seja, diferentes projetistas talvez escolham pontos ligeiramente diferentes do fluxo como pontos limítrofes. De fato, soluções de projeto alternativas podem ser obtidas variando-se o posicionamento das fronteiras do fluxo. Embora deva se tomar cuidado na escolha das fronteiras, uma variação de uma bolha ao longo de uma trajetória de fluxo em geral terá pouco impacto sobre a estrutura de programas final.

As fronteiras dos fluxos para o exemplo são ilustradas na forma de curvas cinza caminhando verticalmente através do fluxo na Figura 9.13. As transformações (bolhas) que constituem o centro de transformações estão entre os dois limites cinza que vão de cima para baixo na figura. Poderia se argumentar a necessidade de reajustar uma fronteira (por exemplo, poderia ser proposta uma fronteira do fluxo de entrada separando *ler sensores* e *obter informações de resposta*). A ênfase nessa etapa do projeto deve ser em selecionar fronteiras razoáveis, e não uma longa iteração para colocação de divisões.

FIGURA 9.13

DFD de nível 3 para monitorar sensores com as fronteiras do fluxo



¹¹ No fluxo de transações, um único dado, denominado *transação*, faz com que o fluxo de dados se ramifique ao longo de uma série de trajetórias de fluxo definida pela natureza da transação.

Passo 5. Realizar a “fatoração de primeiro-nível”. A arquitetura de programas obtida usando-se esse mapeamento resulta em uma distribuição do controle top-down. A *fatoração* conduz a uma estrutura de programas em que os componentes de alto nível tomam as decisões e os componentes de baixo nível realizam a maior parte do trabalho de entrada, cálculos e saída. Os componentes de nível médio realizam parte das tarefas de controle e volumes de trabalho moderados.

Ao se encontrar um fluxo de transformação, um DFD é mapeado em uma estrutura específica (uma arquitetura de chamadas e retornos) que fornece controle para o processamento de informações de entrada, saída e transformação. A fatoração de primeiro nível para o subsistema *monitorar sensores* é ilustrada na Figura 9.14. Um controlador principal (chamado *executivo para monitorar sensores*) reside na parte superior da estrutura de programas e coordena as seguintes funções de controle subordinadas:

- Um controlador de processamento de informações de entrada, denominado *controlador de entrada do sensor*, coordena o recebimento de todos os dados de entrada.
- Um controlador de transformação de fluxo, denominado *controlador de condições de alarme*, supervisiona todas as operações sobre os dados na forma internalizada (por exemplo, um módulo que chama vários procedimentos de transformação de dados).
- Um controlador de processamento de informações de saída, denominado *controlador de saída de alarme*, coordena a produção de informações de saída.

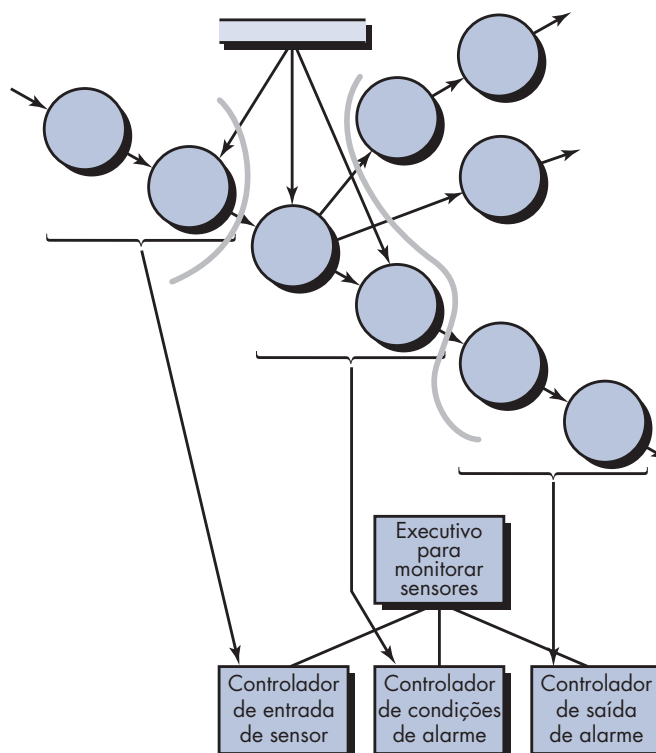
Embora pela Figura 9.14 esteja implícita uma estrutura de três flancos, fluxos complexos em sistemas grandes poderiam ditar dois ou mais módulos de controle para cada uma das funções de controle genéricas descritas anteriormente. O número de módulos no primeiro nível deve ser limitado ao mínimo capaz de realizar funções de controle e ainda assim manter boas características de independência funcional.



Não se torne dogmático neste estágio. Talvez seja necessário estabelecer dois ou mais controladores para cálculo ou processamento de entrada, baseando-se na complexidade do sistema a ser construído. Se o bom senso ditar esta abordagem, faça-a!

FIGURA 9.14

Fatoração de primeiro nível para monitorar sensores





Elimine módulos de controle redundantes. Ou seja, se um módulo de controle não fizer nada mais que controlar um outro módulo, sua função de controle poderia ser implodida para um módulo de nível mais alto.



Mantenha em nível baixo os módulos “trabalhadores” dentro da estrutura de programas. Isso levará a uma arquitetura mais fácil de ser mantida.

Passo 6. Realizar “fatoração de segundo nível”. A fatoração de segundo nível é realizada pelo mapeamento de transformações (bolhas) individuais de um DFD em módulos apropriados dentro da arquitetura. Partindo do limite do centro de transformação e movendo-se para a parte externa ao longo de trajetórias de entrada e depois de saída, as transformações são mapeadas em níveis subordinados da estrutura de software. A abordagem geral da fatoração de segundo nível é ilustrada na Figura 9.15.

Embora a Figura 9.15 ilustre um mapeamento um-para-um entre transformações DFD e módulos de software, frequentemente ocorrem mapeamentos diferentes. Duas ou até mesmo três bolhas podem ser combinadas e representadas como um componente ou uma única bolha poderia ser expandida em dois ou mais componentes. Considerações práticas, bem como medidas de qualidade do projeto, ditam o resultado da fatoração de segundo nível. Revisão e refinamento podem levar a mudanças nessa estrutura, porém pode servir como uma “primeira iteração” de projeto.

A fatoração de segundo nível para fluxo de entrada segue da mesma maneira. A fatoração é mais uma vez realizada movendo-se para fora a partir do limite do centro de transformação ao longo do fluxo de entrada. O centro de transformação do software de subsistema *monitorar sensores* é mapeado de maneira ligeiramente distinta. Cada conversão de dados ou transformação de cálculos da parte de transformação da DFD é mapeada em um módulo subordinado ao controlador de transformação. A Figura 9.16 mostra uma arquitetura de primeira iteração completa.

FIGURA 9.15
Fatoração de segundo nível para monitorar sensores

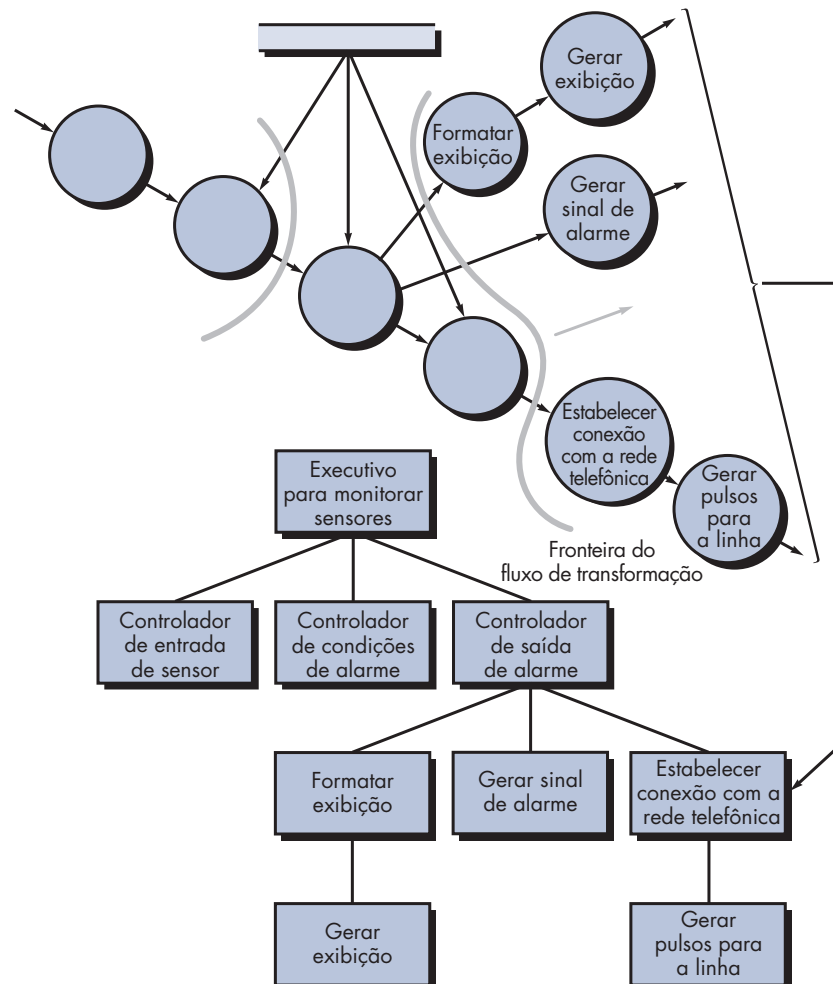
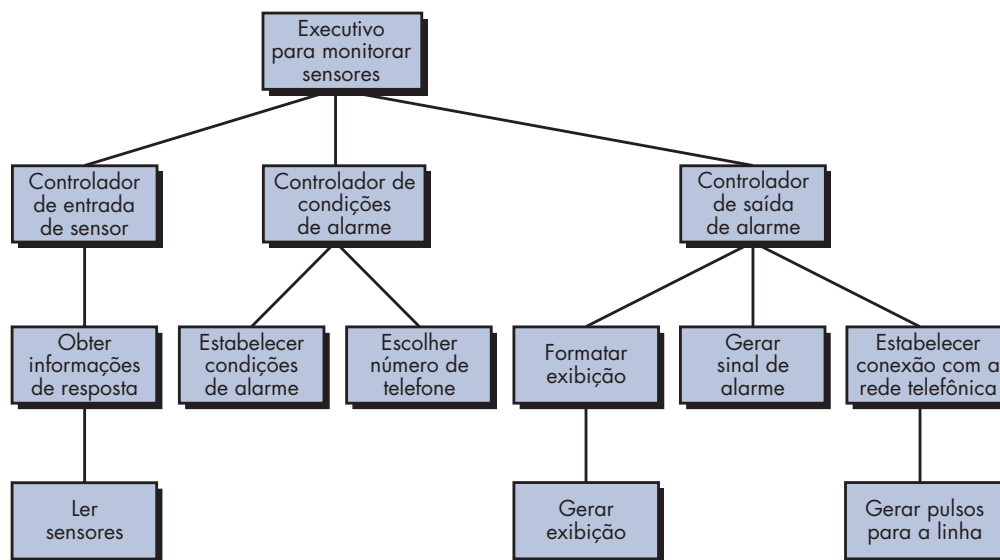


FIGURA 9.16

Estrutura da primeira iteração para monitorar sensores



Os componentes mapeados da maneira anterior e mostrados na Figura 9.16 representam um projeto inicial de arquitetura de software. Embora os componentes sejam chamados para que impliquem função, deve ser redigida uma breve narrativa de processamento (adaptada da especificação de processos desenvolvida para a transformação de dados criada durante a modelagem de requisitos) para cada um deles. A narrativa descreve a interface dos componentes, estruturas de dados internas, uma narrativa funcional e uma breve discussão das restrições e características especiais (por exemplo, entrada/saída de arquivos, características dependentes de hardware, requisitos de timing especiais).

“Faça as coisas da forma mais simples possível. Mas não mais simples do que isso.”

Albert Einstein

Passo 7. Refinar a primeira iteração de arquitetura usando heurística de projeto para aumentar a qualidade do software. Uma primeira iteração de arquitetura sempre pode ser refinada aplicando-se conceitos de independência funcional (Capítulo 8). Componentes são explodidos ou implodidos para produzir a sensata separação de preocupações, coesão adequada, acoplamento mínimo e, o mais importante, uma estrutura que possa ser implementada sem dificuldade, testada sem atribuições e mantida sem aflição.

Os refinamentos são ditados pela análise e métodos de avaliação descritos anteriormente na Seção 9.5, bem como por considerações práticas e bom senso. Há vezes em que, por exemplo, o controlador para fluxo de dados de entrada é completamente desnecessário, quando algum processamento de entrada é necessário em um componente subordinado ao controlador de transformação, quando o alto acoplamento devido a dados globais não pode ser evitado ou quando características estruturais ótimas não podem ser obtidas. Os requisitos de software associados ao discernimento humano são o árbitro final.

O objetivo das sete etapas anteriores é o de desenvolver uma representação de arquitetura de software. Uma vez definida a estrutura, podemos avaliar e refinar a arquitetura de software visualizando-a como um todo. Modificações feitas nessa ocasião requerem pouco trabalho adicional, embora possam ter um impacto profundo na qualidade do software.

Você deve fazer uma pausa por um momento e considerar a diferença entre a abordagem de projeto descrita e o processo de “escrever programas”. Se o código é a única representação de software, você e seus colegas terão grande dificuldade em avaliar ou refinar em um nível global ou holístico e terão, de fato, dificuldade “em distinguir a floresta das árvores”, ou seja, se você ficar muito envolvido com os detalhes não será capaz de ver os pontos importantes como um todo.

CASA SEGURA



Refinamento da arquitetura preliminar

Cena: Sala do Jamie, quando é iniciada a modelagem de projeto.

Atores: Jamie e Ed — membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

[Ed acabou de completar um projeto preliminar do subsistema para monitorar sensores. Ele dá uma passada na sala para pedir a opinião do Jamie.]

Ed: Então, aqui está a arquitetura que obtive. [Ed mostra ao Jamie a Figura 9.16, que ele estuda por alguns instantes.]

Jamie: Excelente, mas acredito que possamos fazer algumas pequenas modificações para torná-la mais simples... E melhor.

Ed: Tais como?

Jamie: Bem, por que você usou o componente *controlador de entrada de sensor*?

Ed: Porque precisamos de um controlador para o mapeamento.

Jamie: Não necessariamente. O controlador não faz muita coisa, já que estamos controlando uma única trajetória de fluxos para os dados de entrada. Podemos eliminar o controlador sem provocar efeitos adversos.

Ed: Tudo bem, sem problemas. Farei a mudança e...

Jamie (sorrindo): Espere! Também podemos implodir os componentes *estabelecer condições de alarme* e *escolher número de telefone*. O controlador de transformações não é realmente necessário e a pequena diminuição na coesão é tolerável.

Ed: Simplificação, hein?

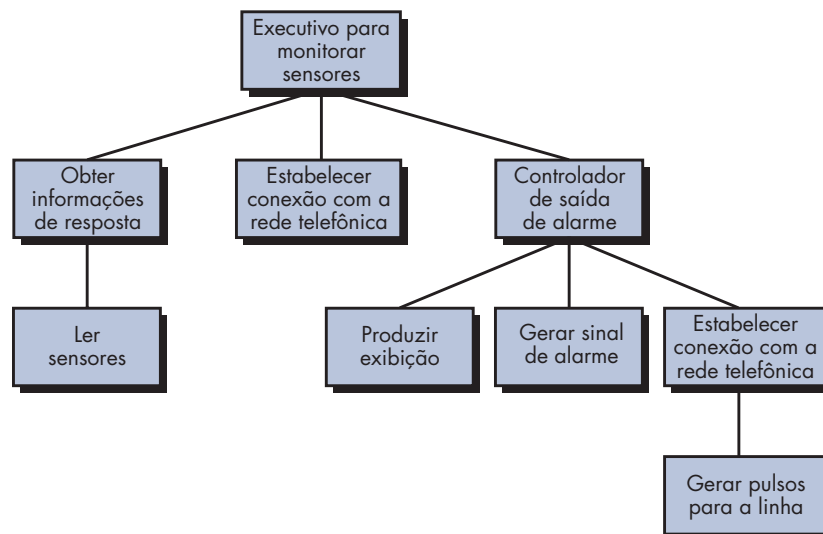
Jamie: Isso mesmo. E enquanto estivermos fazendo os refinamentos, seria uma boa ideia implodir os componentes *formatar exibição* e *gerar exibição*. A formatação de exibição para o painel de controle é simples. Podemos definir um novo módulo chamado *produzir exibição*.

Ed (rascunhando): Então é isso que você acha que deveríamos fazer? [Mostrando ao Jamie a Figura 9.17.]

Jamie: É um bom ponto de partida.

FIGURA 9.17

Estrutura de programas refinada para monitorar sensores



9.6.2 Refinamento do projeto da arquitetura

O que acontece após a arquitetura ter sido criada?

Qualquer discussão de refinamento de projeto deveria ser antecedida pelo seguinte comentário: “Lembre-se de que um ‘projeto ótimo’ que não funciona é de mérito questionável”. Devemos estar preocupados com o desenvolvimento de uma representação de software que atenderá todos os requisitos funcionais e de desempenho e mereça aceitação com base em medidas e heurísticas de projeto.

O refinamento da arquitetura de software durante os estágios iniciais do projeto deve ser encorajado. Conforme discutido neste capítulo, estilos de arquitetura alternativos podem ser obtidos, refinados e avaliados para a “melhor” abordagem. A abordagem para a otimização é um dos verdadeiros benefícios obtidos ao se desenvolver uma representação da arquitetura de software.

É importante notar que a simplicidade estrutural normalmente reflete tanto elegância quanto eficiência. O refinamento de projeto deveria se esforçar por obter o menor número de componentes que seja consistente com efetiva modularidade e a estrutura de dados menos complexa que atenda adequadamente os requisitos de informações.

9.7 RESUMO

A arquitetura de software fornece uma visão holística do sistema a ser construído. Ela representa a estrutura e a organização dos componentes de software, suas propriedades e as conexões entre eles. Os componentes de software incluem módulos de programas e as várias representações de dados manipuladas pelo programa. Consequentemente, o projeto de dados é parte integrante da obtenção da arquitetura de software. A arquitetura destaca decisões de projeto iniciais e fornece um mecanismo para considerar os benefícios de estruturas alternativas do sistema.

Uma série de estilos e padrões de arquitetura encontra-se à disposição do engenheiro de software e pode ser aplicada em um gênero de arquitetura. Cada estilo descreve uma categoria de sistema que engloba um conjunto de componentes que realiza uma função exigida por um sistema; um conjunto de conectores que possibilita a comunicação, coordenação e cooperação entre os componentes; restrições que definem como os componentes podem ser integrados para formar o sistema; e modelos semânticos que permitem a um projetista compreender as propriedades gerais de um sistema.

De modo geral, o projeto da arquitetura é realizado usando-se quatro etapas distintas. Primeiramente, o sistema deve ser representado no contexto. Ou seja, o projeto deve definir as entidades externas com as quais o software interage e a natureza da interação. Uma vez que o contexto tenha sido especificado, o projetista deve identificar um conjunto de abstrações de alto nível, denominado arquétipos, que representam elementos fundamentais do comportamento ou função do sistema. Depois de as abstrações definidas, o projeto começa a se aproximar do domínio da implementação. Os componentes são identificados e representados no contexto de uma arquitetura que os suporta. Por fim, são desenvolvidas instâncias específicas da arquitetura para “provar” o projeto em um contexto do mundo real.

Como um exemplo simples de projeto da arquitetura, o método de mapeamento apresentado neste capítulo usa características de fluxo de dados para obter um estilo de arquitetura comumente usado. Um diagrama de fluxo de dados (DFD) é mapeado em uma estrutura de programas usando uma abordagem de mapeamento de transformação. O mapeamento de transformação é aplicado a um fluxo de informações que apresenta fronteiras distintas entre dados de entrada e saída. O DFD é mapeado em uma estrutura que aloca controle para a entrada, o processamento e a saída ao longo de três hierarquias de módulos fatorados separadamente. Uma vez que a arquitetura tenha sido obtida, ela é elaborada e analisada usando-se critérios de qualidade.

PROBLEMAS E PONTOS A PONDERAR

- 9.1.** Usando a arquitetura de uma casa ou edifício como metáfora, faça comparações com arquitetura de software. Em que sentido as disciplinas da arquitetura clássica e arquitetura de software são similares? Como diferem?
- 9.2.** Apresente dois ou três exemplos de aplicações para cada um dos estilos de arquitetura citados na Seção 9.3.1.
- 9.3.** Alguns dos estilos de arquitetura citados na Seção 9.3.1 são hierárquicos por natureza e outros não. Faça uma lista de cada um dos tipos. Como os estilos de arquitetura que não são hierárquicos seriam implementados?
- 9.4.** Os termos *estilo de arquitetura*, *padrão de arquitetura* e *framework* (não discutido neste livro) são muitas vezes encontrados em discussões sobre arquitetura de software. Pesquise e descreva como cada um deles difere de seus equivalentes.

- 9.5.** Escolha uma aplicação com a qual esteja familiarizado. Responda cada uma das perguntas apresentadas para controle e dados na Seção 9.3.3.
- 9.6.** Pesquise o ATAM (usando [Kaz98]) e apresente uma discussão detalhada das seis etapas indicadas na Seção 9.5.1.
- 9.7.** Caso já não tenha feito, complete o Problema 6.6. Use os métodos de projeto descritos no presente capítulo para desenvolver uma arquitetura de software para a PHTRS.
- 9.8.** Usando um diagrama de fluxo de dados e uma narrativa de processamento, descreva um sistema baseado em computador que tenha características distintas de fluxo de transformações. Defina as fronteiras dos fluxos e mapeie o DFD em uma arquitetura de software usando a técnica descrita na Seção 9.6.1.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A literatura sobre arquitetura de software explodiu ao longo da última década. Livros como os de Gorton (*Essential Software Architecture*, Springer, 2006), Reekie e McAdam (*A Software Architecture Primer*, Angophora Press, 2006), Albin (*The Art of Software Architecture*, Wiley, 2003) e Bass e seus colegas (*Software Architecture in Practice*, 2. ed., Addison-Wesley, 2002) apresentam úteis introduções para uma área intelectualmente desafiadora.

Buschman e seus colegas (*Pattern-Oriented Software Architecture*, Wiley, 2007) e Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004) discutem aspectos orientados a padrões de projeto da arquitetura. Rozanski e Woods (*Software Systems Architecture*, Addison-Wesley, 2005), Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Clements e seus colegas (*Documenting Software Architecture: View and Beyond*, Addison-Wesley, 2002), Bosch [Bos00] e Hofmeister e seus colegas [Hof00] oferecem completos tratados sobre arquitetura de software.

Hennessey e Patterson (*Computer Architecture*, 4. ed., Morgan-Kaufmann, 2007) adotam uma visão quantitativa distinta sobre as questões de projeto da arquitetura de software. Clements e seus colegas (*Evaluating Software Architectures*, Addison-Wesley, 2002) consideram as questões associadas com a avaliação de alternativas de arquitetura e a escolha da melhor arquitetura para determinado domínio de problema.

Livros abordando implementações específicas sobre arquitetura tratam o projeto da arquitetura em uma tecnologia ou ambiente de desenvolvimento específico. Marks e Bell (*Service-Oriented Architecture*, Wiley, 2006) discutem um método de projeto que associa recursos comerciais e computacionais aos requisitos definidos por clientes. Stahl e seus colegas (*Model-Driven Software Development*, Wiley, 2006) discutem a arquitetura no contexto de abordagens de modelagem específicas a um domínio de aplicação. Radaideh e Al-ameed (*Architecture of Reliable Web Applications Software*, GI Global, 2007) consideram arquiteturas apropriadas para WebApps. Clements e Northrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) tratam o projeto de arquiteturas que suportam linhas de produtos de software. Shanley (*Protected Mode Software Architecture*, Addison-Wesley, 1996) fornece uma orientação para projeto da arquitetura para qualquer um que esteja desenvolvendo sistemas operacionais em tempo real baseados em PCs, sistemas operacionais multi-tarefa ou drivers de dispositivos.

A pesquisa atual em arquitetura de software é documentada anualmente nos *Proceedings of the International Workshop on Software Architecture*, patrocinados pela ACM e outras organizações de computação e nos *Proceedings of the International Conference on Software Engineering*.

Uma ampla gama de fontes de informação sobre projeto da arquitetura se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevante para o projeto da arquitetura, pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

acoplamento	268
coesão	266
componentes	
adaptação	281
classificação	282
composição	281
orientados a	
objetos.	260
qualificação	280
tradicionais.	275
WebApp	274
desenvolvimento	
baseado em	
componentes.	279
diretrizes de projeto.	265
engenharia de	
domínio	280
notação tabular	
de projetos.	277
projeto de conteúdo	274

O projeto de componentes ocorre depois de a primeira iteração do projeto da arquitetura tiver sido completa. Neste estágio, a estrutura geral dos dados e programas do software já foi estabelecida. O intuito é traduzir o modelo de projeto em software operacional. Porém, o nível de abstração do modelo de projetos existente é relativamente alto e o nível de abstração do programa operacional é baixo. A tradução pode ser desafiadora, é uma porta aberta para a introdução de erros sutis difíceis de detectar e corrigir em estágios posteriores do processo de software. Em uma famosa palestra, Edsger Dijkstra, um importante contribuidor para nosso entendimento de projeto de software, afirmou [Dij72]:

Software parece ser diferente de muitos outros produtos, em que, via de regra, maior qualidade implica preços mais elevados. Aqueles que realmente querem software confiável descobrirão que devem, primeiro, encontrar um meio de evitar a maioria dos bugs e, como resultado, o processo de programação se tornará mais barato... Programadores eficientes... Não devem perder tempo depurando — e, para início de conversa, não devem introduzir erros.

Embora essas palavras tenham sido proferidas muitos anos atrás, ainda são verdadeiras hoje em dia. Ao traduzirmos o modelo de projetos em código-fonte, devemos seguir um conjunto de princípios de projeto que não apenas realizam a tradução, como também não “introduzam bugs desde o início”.

PANORAMA

O que é? Um conjunto completo de componentes de software é definido durante o projeto da arquitetura. Porém, os detalhes de processamento e estruturas de dados internas de cada componente não são representados em um nível de abstração próximo ao código. O projeto de componentes define as estruturas de dados, os algoritmos, as características das interfaces e os mecanismos de comunicação alocados a cada componente de software.

Quem realiza? Um engenheiro de software realiza o projeto de componentes.

Por que é importante? Você deve ser capaz de determinar se o software irá funcionar ou não antes de construí-lo. O projeto de componentes representa o software de maneira que lhe permita revisar os detalhes do projeto em termos de correção e consistência com outras representações de projeto (isto é, os projetos de interfaces, arquitetura e dados). Ele fornece um meio para avaliar se as estruturas de dados, interfaces e algoritmos vão funcionar.

Quais são as etapas envolvidas? As representações de projeto de dados, arquitetura e interfaces formam a base para o projeto de com-

ponentes. A definição de classes ou a narrativa de processamento para cada um dos componentes é traduzida em um projeto detalhado que faz uso de formas esquemáticas ou baseadas em texto que especificam estruturas de dados internas, detalhes de interfaces locais e lógica de processamento. A notação de projeto engloba diagramas UML e representações complementares. O projeto procedural é especificado usando-se um conjunto de construções da programação estruturada. Normalmente é possível adquirirem-se componentes de software reutilizáveis em vez de construir novos componentes.

Qual é o artefato? O projeto para cada componente, representado em notação gráfica, tabular ou baseada em texto, é o principal artefato durante o projeto de componentes.

Como garantir que o trabalho foi realizado corretamente? Uma revisão do projeto é realizada. O projeto é examinado para determinar se as estruturas de dados, interfaces, sequências de processamento e condições lógicas estão corretas e irão produzir a transformação de controle ou de dados apropriada atribuída ao componente durante etapas anteriores do projeto.

É possível representar o projeto de componentes usando uma linguagem de programação. Em essência, o programa é criado por meio do modelo de projeto da arquitetura como diretriz. Uma abordagem alternativa é representar o projeto de componentes usando alguma representação intermediária (por exemplo, gráfica, tabular ou baseada em texto) que possa ser traduzida facilmente em código-fonte. Independentemente do mecanismo utilizado para representar o projeto de componentes, as estruturas de dados, as interfaces e os algoritmos definidos devem atender uma série de diretrizes de projeto bem fundamentadas que auxiliam a evitar erros à medida que o projeto procedural evolui. Neste capítulo, examinaremos essas diretrizes de projeto e os métodos disponíveis para atingi-las.

10.1 O QUE É COMPONENTE?

“Detalhes não são detalhes. Eles fazem o projeto.”

Charles Eames

Componente é um bloco construtivo modular para software de computador. Mais formalmente, a Especificação da Linguagem de Modelagem Unificada da OMG (*OMG Unified Modeling Language Specification* [OMG03a]) define componente como “... Uma parte modular, possível de ser implantada e substituível de um sistema que encapsula implementação e expõe um conjunto de interfaces”.

Conforme discutido no Capítulo 9, os componentes preenchem a arquitetura de software e, como consequência, desempenham um papel para alcançar os objetivos e requisitos do sistema a ser construído. Pelo fato de os componentes residirem na arquitetura de software, devem se comunicar e colaborar com outros componentes e entidades (por exemplo, outros sistemas, dispositivos, pessoas) existentes fora dos limites do software.

O verdadeiro significado do termo *componente* diferirá dependendo do ponto de vista do engenheiro de software que o utiliza. Nas seções seguintes, examinaremos três importantes visões do que é um componente e como é utilizado à medida que a modelagem de projetos prossegue.

10.1.1 Uma visão orientada a objetos

PONTO-CHAVE

Sob o ponto de vista orientado a objetos, um componente é um conjunto de classes colaborativas.

No contexto da engenharia de software orientada a objetos, um componente contém um conjunto de classes colaborativas.¹ Cada classe contida em um componente foi completamente elaborada para incluir todos os atributos e operações relevantes à sua implementação. Como parte da elaboração do projeto, todas as interfaces que permitem que as classes se comuniquem e colaborem com outras classes de projeto também precisam ser definidas. Para tanto, começamos com o modelo de requisitos e elaboramos as classes de análise (para componentes que se relacionam com o domínio do problema), bem como as classes de infraestrutura (para componentes que oferecem suporte a serviços para o domínio do problema).

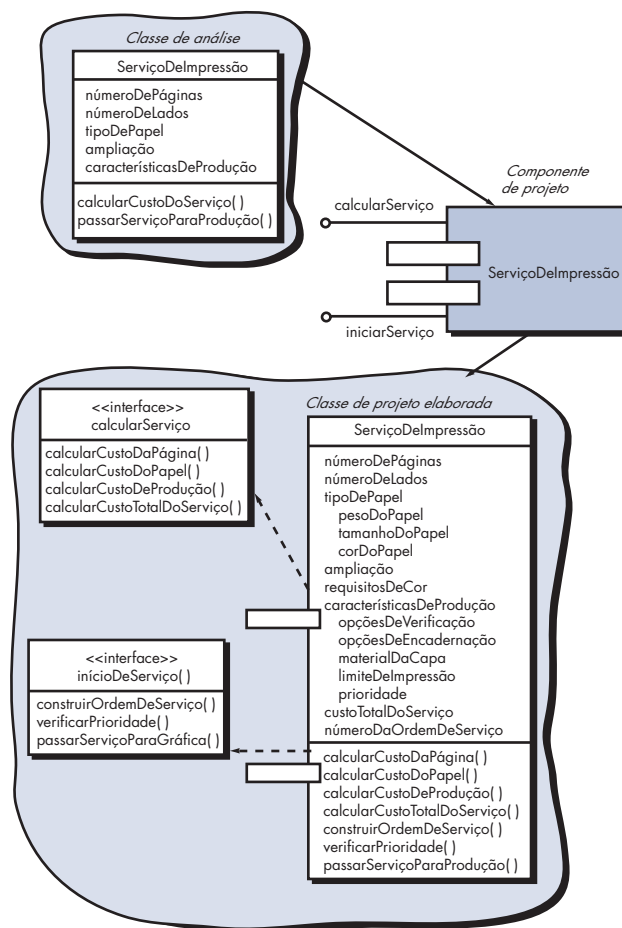
Para ilustrarmos o processo de elaboração de projeto, consideremos o software a ser criado para uma sofisticada gráfica. O objetivo geral do software é coletar os requisitos do cliente na recepção da loja, orçar um trabalho e, em seguida, passar a tarefa para um centro de produção automatizado. Durante a engenharia de requisitos, foi obtida uma classe de análise denominada **ServiçoDeImpressão**. Os atributos e operações definidos durante a análise são indicados na parte superior da Figura 10.1. Durante o projeto da arquitetura, **ServiçoDeImpressão** é definida como um componente na arquitetura de software e é representada usando notação abreviada² UML no centro à direita da figura. Observe que **ServiçoDeImpressão** possui duas interfaces, *calcularServiço*, que fornece capacidade para orçar um trabalho e *iniciarServiço*, que passa adiante a tarefa para o centro de produção. Estas são representadas usando-se os símbolos de “pirulito” exibidos à esquerda do retângulo do componente.

¹ Em alguns casos, um componente pode conter uma única classe.

² Os leitores que não estiverem familiarizados com a notação UML devem consultar o Apêndice 1.

FIGURA 10.1

Elaboração de um componente de projeto



Lembre-se de que tanto a modelagem da análise quanto a modelagem do projeto são ações iterativas. Elaborar a classe de análise original poderia exigir etapas de análise adicionais, então seguidas de etapas de modelagem de projetos para representar a classe de projeto elaborada (os detalhes do componente).

O projeto de componentes se inicia neste ponto. Os detalhes do componente **ServiçoDeImpressão** devem ser elaborados para fornecer informações suficientes para orientar a implementação. A classe de análise original é elaborada para dar corpo a todos os atributos e operações necessárias para implementar a classe na forma do componente **ServiçoDeImpressão**. Referindo-se à parte inferior direita da Figura 10.1, a classe de projeto elaborada, **ServiçoDeImpressão**, contém informações de atributos mais detalhadas, bem como uma descrição expandida das operações necessárias para implementar o componente. As interfaces `calcularServiço` e `iniciarServiço` implicam comunicação e colaboração com outros componentes (não indicados aqui). Por exemplo, a operação `calcularCustoDaPágina()` (parte da interface `cálculoDeServiço`) poderia colaborar com um componente **TabelaDePreço** contendo informações sobre os preços de serviços. A operação `verificarPrioridade()` (parte da interface `inícioDeServiço`) poderia colaborar com um componente **FilaDeServiço** para determinar os tipos e prioridades dos serviços atualmente em espera para produção.

A atividade de elaboração é aplicada a todos os componentes definidos como parte do projeto da arquitetura. Uma vez completada, aplica-se uma maior elaboração a cada atributo, operação e interface. As estruturas de dados apropriadas para cada atributo devem ser especificadas. Além disso, o detalhe algorítmico exigido para implementar a lógica de processamento associada a cada operação é desenvolvido. Essa atividade de projeto procedural é discutida posteriormente, ainda neste capítulo. Por fim, desenvolvem-se os mecanismos necessários para implementar a

interface. Para software orientado a objetos, isso poderia englobar a descrição de todas as mensagens necessárias para efetivar a comunicação entre objetos de um sistema.

10.1.2 A visão tradicional

No contexto da engenharia de software tradicional, componente é o elemento funcional de um programa que incorpora a lógica de processamento, as estruturas de dados internas necessárias para implementar a lógica de processamento e uma interface que habilita o componente a ser chamado e que dados sejam passados a ele. Um componente tradicional, também denominado *módulo*, reside na arquitetura de software e se presta a um de três importantes papéis: (1) um *componente de controle* que coordena a chamada de todos os demais componentes do domínio do problema, (2) um *componente de domínio do problema* que implementa uma função completa ou parcial solicitada pelo cliente ou (3) um *componente de infraestrutura* responsável por funções que dão suporte ao processamento necessário no domínio do problema.

Assim como os componentes orientados a objetos, os componentes de software tradicionais são obtidos do modelo de análise. Nesse caso, entretanto, o elemento orientado a fluxos de dados do modelo de análise serve como base para essa obtenção. Cada transformação (bolha), representada nos níveis mais baixos do diagrama de fluxo de dados, é mapeada em uma hierarquia de módulos. Os componentes de controle (módulos) residem próximo do alto da hierarquia (arquitetura de programas), e os componentes de domínio do problema tendem a residir mais próximo da parte inferior da hierarquia. Para alcançar modularidade efetiva, conceitos de projeto como independência funcional (Capítulo 8) são aplicados durante a elaboração dos componentes.

Para ilustrarmos o processo de elaboração de projetos para componentes tradicionais, consideremos novamente o software a ser criado para uma gráfica sofisticada. Um conjunto de diagramas de fluxo de dados seria obtido durante a modelagem de requisitos. Suponhamos que estas sejam mapeadas na arquitetura da Figura 10.2. Cada retângulo representa um componente de software. Observe que os retângulos sombreados são equivalentes, em termos funcionais, às operações definidas para a classe **ServiçoDeImpressão** discutidas na Seção 10.1.1. Nesse caso, entretanto, cada operação é representada como um módulo separado chamado conforme indicado na figura. São usados outros módulos para controlar o processamento, e estes são, portanto, componentes de controle.

Durante o projeto de componentes, cada módulo da Figura 10.2 é elaborado. A interface de módulos é definida explicitamente. Ou seja, cada objeto de dados ou de controle que flui através da interface é representado. São definidas as estruturas de dados utilizadas internamente ao módulo. O algoritmo que possibilita ao módulo cumprir sua função é desenhado usando-se o método de refinamento gradual discutido no Capítulo 8. Algumas vezes o comportamento do módulo é representado usando-se um diagrama de estado.

Para ilustrarmos o processo, consideremos o módulo *CalcularCustoPorPágina*. O intuito é calcular o custo de impressão por página tomando como base as especificações fornecidas pelo cliente. Os dados necessários para realizar essa função são: o **número de páginas contidas no documento**, o **número total de documentos a ser produzidos**, **impressão frente ou frente e verso**, **requisitos de cores**, **requisitos de tamanho**. Esses dados são passados para *CalcularCustoPorPágina* através da interface do módulo. *CalcularCustoPorPágina* usa-os para determinar o custo de uma página baseado no tamanho e na complexidade do trabalho — uma função de todos os dados passados para o módulo via interface. O custo por página é inversamente proporcional ao tamanho do serviço e diretamente proporcional à complexidade do serviço.

A Figura 10.3 representa o projeto de componentes usando uma notação UML modificada. O módulo *CalcularCustoPorPágina* acessa dados chamando o módulo *obterDadosDoServiço*, que possibilita que todos os dados relevantes sejam passados para o componente e uma interface de banco de dados, *acessarBDCustos*, que possibilita que o módulo acesse um banco de dados contendo todos os custos de impressão. À medida que o projeto continua, o módulo

“Invariavelmente, constata-se que um sistema complexo que funciona é a evolução de um sistema simples que funcionava.”

John Gall



À medida que o projeto para cada componente de software é elaborado, o foco passa para o projeto de estruturas de dados específicas e projeto procedural para manipular as estruturas de dados. Entretanto, não se esqueça da arquitetura que deve abrigar os componentes ou as estruturas de dados globais que poderão atender vários componentes.

FIGURA 10.2

Diagrama de estruturas para um sistema tradicional

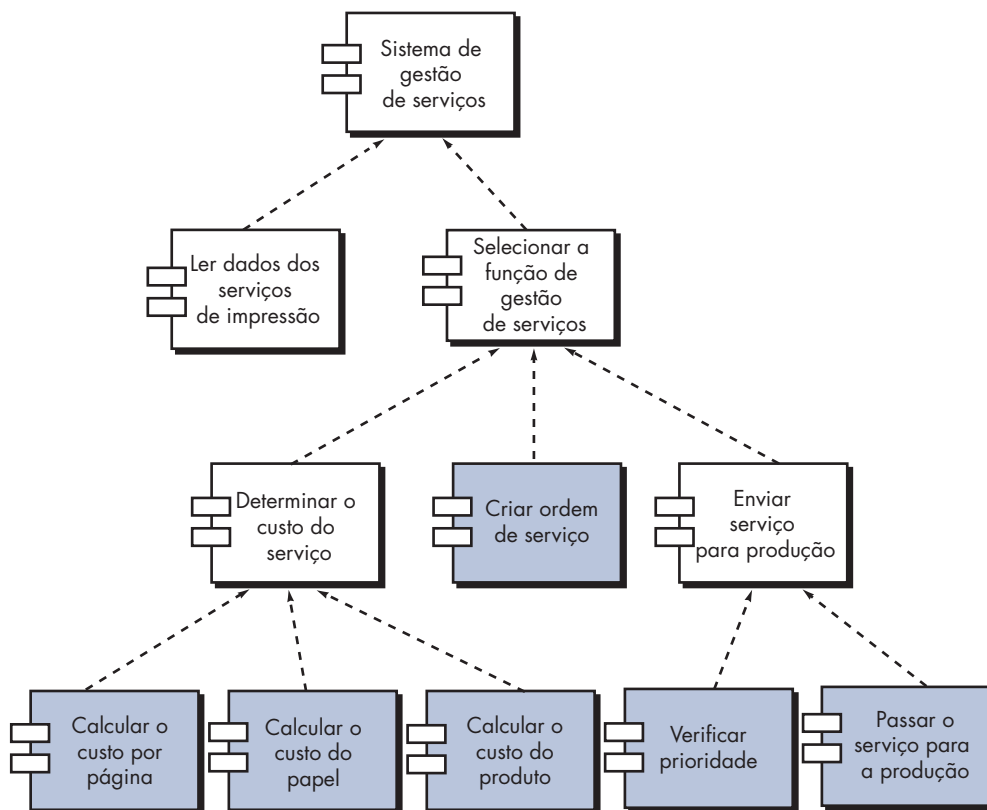
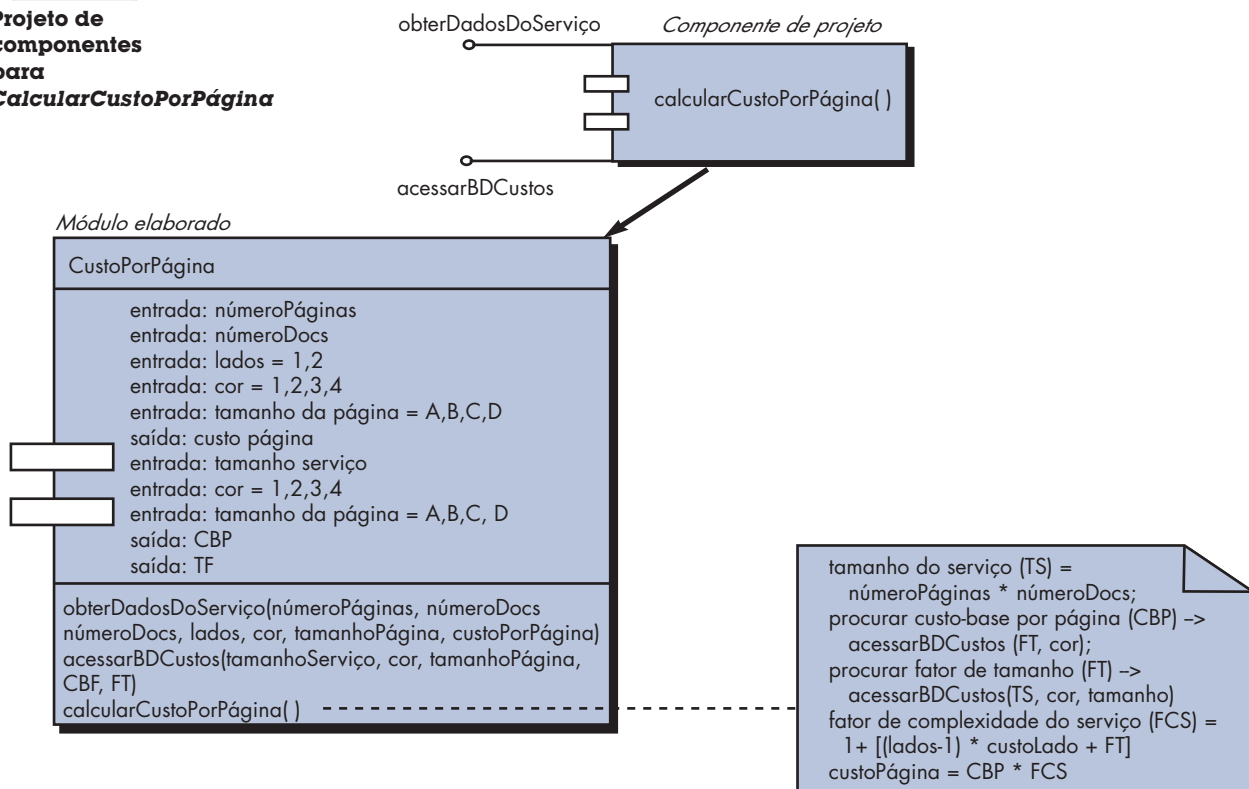


FIGURA 10.3

Projeto de componentes para *CalcularCustoPorPágina*



CalcularCustoPorPágina é elaborado para fornecer detalhes algorítmicos e detalhes de interface (Figura 10.3). Os detalhes algorítmicos podem ser representados usando-se o texto em pseudocódigo mostrado na figura ou por meio de um diagrama de atividades UML. As interfaces são representadas como um conjunto de itens ou objetos de dados de entrada/saída. A elaboração de projeto continua até que detalhes suficientes tenham sido fornecidos para orientar a construção do componente.

10.1.3 Uma visão relacionada com processos

As visões tradicionais e orientadas a objetos de projeto de componentes apresentadas nas Seções 10.1.1 e 10.1.2 partem do pressuposto de que o componente está sendo desenhado a partir do zero. Isto é, temos de criar um componente baseado nas especificações obtidas do modelo de requisitos. Existe, obviamente, uma outra abordagem. Ao longo das duas últimas décadas, a comunidade de engenharia de software tem enfatizado a necessidade de se construírem sistemas que façam uso de componentes de software ou de padrões de projeto existentes. Em essência, é colocado à disposição dos profissionais de software um catálogo de projetos ou código de componentes de qualidade comprovada à medida que o trabalho de projeto prossegue. À medida que a arquitetura de software é desenvolvida, escolhemos componentes ou padrões de projeto desse catálogo e os usamos para preencher a arquitetura. Pelo fato de esses componentes terem sido criados tendo a reusabilidade em mente, uma descrição completa de suas interfaces, a(s) função(ões) por eles realizada(s) e a comunicação e colaboração por eles exigidas estarão à nossa disposição. Discutiremos posteriormente, na Seção 10.6, alguns dos importantes aspectos da engenharia de software baseada em componentes (*component-based software engineering*, CBSE).



Padrões e estruturas baseadas em componentes

Um dos elementos-chave que levam ao sucesso ou insucesso da CBSE é a disponibilidade de padrões baseados em componentes, algumas vezes denominados *middleware*. *Middleware* é um conjunto de componentes de infraestrutura que possibilita que os componentes do domínio do problema se comuniquem entre si através de uma rede ou em um sistema complexo. Os engenheiros de software que quiserem usar desenvolvimento baseado em componentes como processo de software poderão escolhê-los entre os seguintes padrões:

OMG CORBA — www.corba.org/

Microsoft COM — www.microsoft.com/com/tech/complus.asp

Microsoft .NET — <http://msdn2.microsoft.com/en-us/netframework/default.aspx>

Sun JavaBeans — <http://java.sun.com/products/ejb/>

Os sites citados apresentam uma ampla gama de tutoriais, artigos, ferramentas e recursos gerais sobre os importantes padrões de *middleware*.

INFORMAÇÕES

10.2 PROJETO DE COMPONENTES BASEADOS EM CLASSES

Conforme já citado, o projeto de componentes apoia-se nas informações desenvolvidas como parte do modelo de requisitos (Capítulos 6 e 7) e representadas como parte do modelo da arquitetura (Capítulo 9). Quando é escolhida uma abordagem de engenharia de software orientada a objetos, o projeto de componentes focaliza a elaboração de classes específicas do domínio do problema e a definição e o refinamento de classes de infraestrutura contidas no modelo de requisitos. A descrição detalhada dos atributos, operações e interfaces utilizados por essas classes é o detalhe de projeto requerido como precursor da atividade de construção.

10.2.1 Princípios básicos de projeto

Quatro princípios básicos de projeto são aplicáveis ao projeto de componentes e têm sido largamente adotados quando se aplica a engenharia de software orientada a objetos. A motivação por trás da aplicação desses princípios é criar projetos que sejam mais fáceis de modificar

e reduzir a propagação de efeitos colaterais na ocorrência de modificações. Podemos usar tais princípios como guia à medida que cada componente de software é desenvolvido.

Princípio do aberto-fechado (Open-Closed Principle, OCP). “Um módulo [componente] deve ser aberto para a extensão, mas fechado para modificações” [Mar00]. Essa afirmação pode parecer uma contradição, mas representa uma das características mais importantes de um bom projeto de componentes. Em outras palavras, devemos especificar o componente para permitir que ele seja estendido (em seu domínio funcional) sem a necessidade de fazer modificações internas (em nível de código ou lógica) no próprio componente. Para tanto, criamos abstrações que servem como um buffer entre a funcionalidade que provavelmente será estendida e a classe de projeto em si.

Por exemplo, suponhamos que a função de segurança do *CasaSegura* faça uso de uma classe **Detector** que deve verificar o estado de cada tipo de sensor de segurança. É provável que à medida que o tempo for passando, o número e os tipos de sensores de segurança crescerão. Se a lógica de processamento interna for implementada como uma sequência de construtos *se-então-senão*, cada um deles tratando de um tipo de sensor diferente, a adição de um novo tipo de sensor exigirá lógica de processamento interna adicional (ainda um outro *se-então-senão*). Essa é uma violação do OCP.

Uma forma de concretizar o OCP para a classe **Detector** é ilustrada na Figura 10.4. A interface *sensor* apresenta uma visão consistente dos sensores para o componente detector. Se for adicionado um novo tipo de sensor, nenhuma mudança é necessária na classe **Detector** (componente). O OCP é preservado.

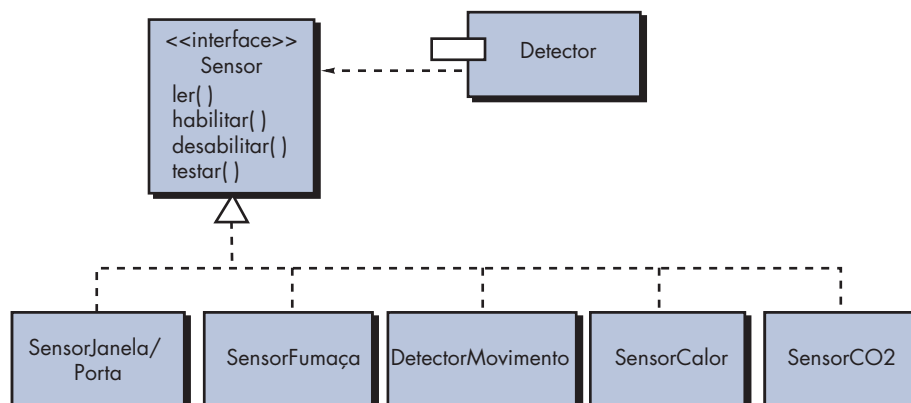
O princípio da substituição de Liskov (LSP). “As subclasses devem ser substitutas de suas classes-base” [Mar00]. Esse princípio de projeto, originalmente proposto por Barbara Liskov [Lis88], sugere que um componente que usa uma classe-base deveria continuar a funcionar apropriadamente caso uma classe derivada da classe-base fosse passada para o componente em seu lugar. O LSP exige que qualquer classe derivada de uma classe-base deva honrar qualquer contrato implícito entre a classe-base e os componentes que a usam. No contexto dessa discussão, um “contrato” é uma *precondição* que deve ser verdadeira antes de o componente usar uma classe-base e uma *pós-condição* que deve ser verdadeira após o componente usar uma classe-base. Ao criar classes derivadas, certifique-se de que atendem às pré e pós-condições.

Princípio da inversão da dependência (DIP). “Dependa de abstrações. Não dependa de concretizações” [Mar00]. Como vimos na discussão sobre OCP, abstrações é o lugar onde um projeto pode ser estendido sem grandes complicações. Quanto mais um componente depender de outros componentes concretos (e não de abstrações como uma interface), mais difícil será estendê-lo.



Se você dispensa o projeto e vai direto ao código, lembre-se apenas de que código é a “concretização” final. Você estará violando o DIP.

FIGURA 10.4
Seguindo o OCP



CASA SEGURA

**O OCP em ação**

Cena: Sala do Vinod.

Atores: Vinod e Shakira — membros da equipe de engenharia de software do CasaSegura.

Conversa:

Vinod: Acabei de receber uma ligação do Doug [o gerente da equipe]. Ele me disse que o pessoal de marketing quer acrescentar um novo sensor.

Shakira (com um sorriso de superioridade): Outra vez não!

Vinod: Isso mesmo... E você não vai acreditar no que esses caras inventaram.

Shakira: Qual é a surpresa agora?

Vinod (sorrindo): Algo que eles chamaram de sensor de angústia de uau-uau.

Shakira: O quê... ?

Vinod: Destina-se a pessoas que deixam seus bichinhos de estimação em apartamentos ou condomínios ou casas que são próximas uma das outras. O cachorro começa a latir. O vizinho se irrita e reclama. Com esse sensor, se o cachorro latir por, digamos, mais de um minuto, o sensor ativa um modo de alarme especial que chama o dono no seu celular.

Shakira: Você tá brincando, não é?

Vinod: Não. Doug quer saber quanto tempo levará para acrescentar essa característica à função de segurança.

Shakira (pensando um pouco): Não muito... Veja. [Ela mostra ao Vinod a Figura 10.4] Isolamos as verdadeiras classes de sensores por trás da interface **sensor**. Desde que tenhamos as especificações para o sensor de cachorros, acrescentá-lo deve ser moleza. A única coisa que terei de fazer é criar um componente apropriado... Quer dizer, classe, para ele. Nenhuma mudança no componente **Detector** em si.

Vinod: Então, vou dizer ao Doug que não é nada do outro mundo.

Shakira: Conhecendo o Doug, ele nos manterá focados e não irá liberar essa coisa até a próxima versão.

Vinod: Isso não é ruim, mas você consegue implementá-lo agora se ele quiser?

Shakira: Sim, o jeito que projetamos a interface me permite fazê-lo sem grandes complicações.

Vinod (pensando um pouco): Você já ouviu falar do princípio do aberto-fechado?

Shakira (encolhendo os ombros.): Nunca ouvi falar.

Vinod (sorrindo): Sem problemas.

Princípio da segregação de interfaces (ISP). *“É melhor usar várias interfaces específicas de clientes do que uma única interface de propósito geral”* [Mar00]. Há várias ocasiões em que componentes para vários clientes usam uma operação fornecida por uma classe-servidora. O ISP sugere a criação de uma interface especializada para atender cada categoria principal de clientes. Apenas as operações que forem relevantes a determinada categoria de clientes devem ser especificadas na interface para esse cliente. Se vários clientes precisarem das mesmas operações, estas devem ser especificadas em cada uma das interfaces especializadas.

Como um exemplo, considere a classe **Planta** usada para as funções de segurança e vigilância do *ClasseSegura* (Capítulo 6). Para as funções de segurança, **Planta** é usada apenas durante atividades de configuração e usa as operações *colocarDispositivo()*, *mostrarDispositivo()*, *agruparDispositivo()* e *removerDispositivo()* para inserir, mostrar, agrupar e remover sensores da planta. A função de vigilância do *CasaSegura* usa as quatro operações citadas para segurança, porém, também requer operações especiais para gerenciar câmeras: *mostrarFOV()* e *mostrarIDDispositivo()*. Portanto, o ISP sugere que componentes clientes das duas funções do *CasaSegura* tenham interfaces especializadas definidas para eles. A interface para segurança englobaria apenas as operações *colocarDispositivo()*, *mostrarDispositivo()*, *agruparDispositivo()* e *removerDispositivo()*. A interface para vigilância incorporaria as operações *colocarDispositivo()*, *mostrarDispositivo()*, *agruparDispositivo()* e *removerDispositivo()*, juntamente com *mostrarFOV()* e *mostrarIDDispositivo()*.

Embora os princípios de projeto de componentes sejam úteis em termos de orientação, os componentes em si não vivem de forma isolada. Em muitos casos, componentes ou classes individuais são organizados em subsistemas ou pacotes. É razoável perguntar como deve ocorrer a atividade de empacotamento. Exatamente de que forma os componentes devem ser organizados à medida que o projeto prossegue? Martin [Mar00] sugere outros princípios de empacotamento aplicáveis ao projeto de componentes:

PONTO-CHAVE

Projetar componentes para reutilização requer mais que um bom projeto técnico. Essa atividade também requer mecanismos de controle de configuração efetivos (Capítulo 22).

Princípio da equivalência de reuso de versões (REP). *“A granularidade da reutilização é a granularidade da versão”* [Mar00]. Quando as classes ou os componentes são projetados tendo em vista o reuso, existe um contrato implícito estabelecido entre o desenvolvedor da entidade reutilizável e quem irá usá-la. O desenvolvedor se compromete a estabelecer um sistema de controle de versões que ofereça suporte e manutenção a versões mais antigas da entidade enquanto os usuários gradualmente irão atualizando para a versão mais atual. Em vez de tratar cada uma dessas classes individualmente, em geral é recomendável agrupar classes reutilizáveis em pacotes que possam ser gerenciados e controlados à medida que versões mais recentes evoluam.

Princípio do fechamento comum (CCP). *“Classes que mudam juntas, devem ficar juntas”* [Mar00]. As classes devem ser empacotadas de forma coesa. Ao serem empacotadas como parte de um projeto, devem tratar a mesma área funcional ou comportamental. Quando alguma característica dessa área tiver de mudar, é provável que apenas aquelas classes contidas no pacote precisarão ser modificadas. Isso leva a um controle de mudanças e gerenciamento de versões mais efetivo.

Princípio comum da reutilização (CRP). *“As classes que não são reutilizadas juntas não devem ser agrupadas juntas”* [Mar00]. Quando uma ou mais classes contidas em um pacote muda(m), o número da versão do pacote muda. Todas as demais classes ou pacotes que dependem do pacote alterado agora precisam ser atualizadas para a versão mais recente do pacote e serem testadas para garantir que a nova versão opere sem incidentes. Se as classes não forem agrupadas de forma coesa, é possível que uma classe sem nenhuma relação com as demais contidas em um pacote seja alterada. Isso precipitará integração e testes desnecessários. Por essa razão, apenas as classes reutilizadas juntas devem ser incluídas em um pacote.

10.2.2 Diretrizes para o projeto de componentes

Além dos princípios discutidos na Seção 10.2.1, pode-se aplicar um conjunto de diretrizes de projeto pragmáticas à medida que o projeto de componentes prossegue. Tais diretrizes se aplicam a componentes, suas interfaces e às características de dependência e herança que têm algum impacto sobre o projeto resultante. Ambler [Amb02b] sugere as seguintes diretrizes:

Componentes. Devem-se estabelecer convenções de nomes para componentes especificados como parte do modelo de arquitetura e, então, refiná-los e elaborá-los como parte do modelo de componentes. Os nomes de componentes de arquitetura devem ser extraídos do domínio do problema e devem ter significado para todos os interessados que veem o modelo da arquitetura. Por exemplo, o nome de classe **Planta** é significativo para qualquer um que o leia, independentemente de sua bagagem técnica. Por outro lado, componentes de infraestrutura ou classes elaboradas no nível de componentes devem receber nomes que reflitam significados específicos à implementação. Se uma lista ligada tiver de ser gerenciada como parte da implementação **Planta**, a operação *gerenciarLista()* é apropriada, mesmo que uma pessoa não técnica possa interpretá-la de forma incorreta.³ Podemos optar pelo uso de estereótipos para auxiliar na identificação da natureza dos componentes no nível de projeto mais detalhado. Por exemplo, poderíamos usar **<<infrastructure>>** para identificar um componente de infraestrutura, **<<database>>** para identificar um banco de dados que atenda uma ou mais classes de projeto ou o sistema inteiro; **<<table>>** poderia ser usado para identificar uma tabela em um banco de dados.

Interfaces. As interfaces nos dão importantes informações sobre a comunicação e a colaboração (bem como nos ajuda a alcançar o OCP). Entretanto, a representação totalmente livre de interfaces tende a complicar os diagramas de componentes. Ambler [Amb02c] recomenda o seguinte: (1) a representação “pirulito” de uma interface deve ser usada em conjunto com a abordagem mais formal da UML que usa retângulos e setas pontilhadas, quando os diagramas se tornarem mais complexos; (2) para consistência, as interfaces devem fluir da esquerda para a direita do

 O que devemos considerar ao atribuímos nomes aos componentes?

³ É pouco provável que alguém de marketing ou da empresa do cliente (uma pessoa não técnica) examine informações detalhadas de projeto.

retângulo de componente; (3) devem ser mostradas apenas aquelas interfaces relevantes para o componente em consideração, mesmo que estejam disponíveis outras interfaces. Tais recomendações destinam-se a simplificar a natureza visual dos diagramas de componentes da UML.

Dependências e herança. Para melhorar a legibilidade, é aconselhável modelar as dependências da esquerda para a direita e as heranças de baixo (classes derivadas) para cima (classes-base). Além disso, as interdependências de componentes devem ser representadas através de interfaces, e não por meio da representação de uma dependência componente-para-componente. Seguindo a filosofia do OCP, isso facilitará a manutenção do sistema.

10.2.3 Coesão

No Capítulo 8, descrevemos coesão como o “foco único” de um componente. No contexto do projeto de componentes para sistemas orientados a objetos, *coesão* implica um componente ou classe encapsular apenas atributos e operações que estejam intimamente relacionados entre si e com a classe ou o componente em si. Lethbridge e Laganière [Let01] definem uma série de tipos diferentes de coesão (enumerados em ordem de nível de coesão⁴):

Funcional. Apresentada basicamente por operações, este nível de coesão ocorre quando um componente realiza um cálculo planejado e depois retorna um resultado.

De camadas. Apresentada por pacotes, componentes e classes, esse tipo de coesão ocorre quando uma camada mais alta acessa os serviços de uma camada mais baixa, porém as camadas mais baixas não acessam as mais elevadas. Consideremos, por exemplo, que a função de segurança do *CasaSegura* precise fazer uma ligação telefônica caso um alarme seja acionado. Seria possível definirmos um conjunto de pacotes em camadas conforme mostra a Figura 10.5. Os pacotes sombreados contêm componentes de infraestrutura. O acesso é do pacote de painel de controle para baixo.

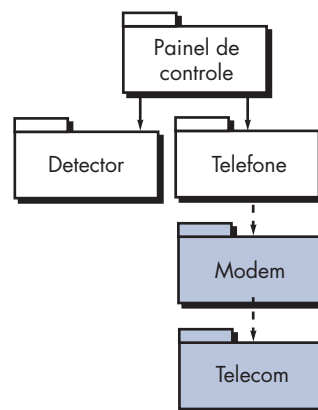
De comunicação. Todas as operações que acessam os mesmos dados são definidas em uma classe. Em geral, tais classes enfocam exclusivamente os dados em questão, acessando-os e armazenando-os.

As classes e componentes que apresentam coesão funcional, de camadas e comunicação são relativamente fáceis de ser implementados, testados e mantidos. Devemos nos esforçar ao máximo para atingir esses níveis de coesão sempre que possível. É importante notar, entretanto, que questões pragmáticas de projeto e implementação algumas vezes nos forçam a optar por níveis de coesão mais baixos.



Embora a compreensão dos vários níveis de coesão seja instrutiva, é mais importante estar atento ao conceito geral ao se projetarem componentes. Mantenha o nível de coesão o mais elevado possível.

FIGURA 10.5
Coesão de camadas



⁴ Em geral, quanto maior o nível de coesão, mais fácil é implementar, testar e manter um componente.

CASA SEGURA

**Coesão em ação**

Cena: Sala do Jamie.

Atores: Jamie e Ed — membros da equipe de engenharia de software do CasaSegura que estão trabalhando na função de vigilância.

Conversa:

Ed: Tenho um projeto preliminar do componente **câmera**.

Jamie: Quer fazer uma rápida revisão?

Ed: Creio que sim... Mas, na realidade, gostaria de sua opinião sobre algo.

(Jamie gesticula para que ele continue.)

Ed: Originalmente definimos cinco operações para **câmera**.

Veja...

`determinarTipo()` informa o tipo de câmera.

`traduzirLocalização()` possibilita que a câmera seja movimentada pela planta.

`exibirID()` obtém o ID da câmera e o exibe próximo do ícone câmera.

`exibirVisão()` mostra graficamente o campo de visão da câmera.

`exibirZoom()` mostra a ampliação da câmera graficamente.

Ed: Projetei cada uma delas separadamente e são operações bastante simples. Portanto, imagino que seria uma boa ideia combinar todas as operações de exibição em apenas uma chamada `exibirCâmera()` — ela mostrará o ID, a vista e a ampliação. O que você acha?

Jamie (fazendo uma careta): Não tenho certeza de que seja uma ideia tão boa assim.

Ed (franzindo a testa): O motivo é que todas essas pequenas operações poderão nos dar dor de cabeça.

Jamie: O problema em combiná-las é perdermos coesão, sabe, a operação `exibirCâmera()` não será focada.

Ed (ligeiramente exasperado): E daí? O conjunto todo terá menos de 100 linhas de código-fonte, no máximo. Será mais fácil implementá-lo, eu acho.

Jamie: E se o Marketing decidir mudar a maneira pela qual representamos o campo de visão?

Ed: Simplesmente mexo na operação `displayCâmera()` e faço a modificação.

Jamie: E os efeitos colaterais?

Ed: O que você quer dizer com isso?

Jamie: Bem, digamos que você faça a modificação, mas, inadvertidamente, crie um problema na exibição do ID.

Ed: Eu não seria tão descuidado assim.

Jamie: Talvez não, mas e se alguém do suporte tiver que, daqui a dois anos, fazer a modificação. Pode ser que não entenda a operação tão bem quanto você e, vai saber, talvez ele seja descuidado.

Ed: Então você é contrário a isso?

Jamie: Você é o projetista... A decisão é sua... Apenas certifique-se de ter compreendido as consequências da baixa coesão.

Ed (refletindo um pouco): Talvez seja melhor mesmo fazer duas operações de exibição separadas.

Jamie: Ótima decisão.

10.2.4 Acoplamento

À medida que o projeto para cada componente de software é elaborado, o foco passa para o projeto de estruturas de dados específicas e para o projeto procedural para a manipulação de estruturas de dados. Entretanto, não se esqueça da arquitetura que deve abrigar os componentes ou as estruturas de dados globais que podem servir vários componentes.

Em discussões anteriores sobre análise e projeto, observamos que a comunicação e a colaboração são elementos essenciais de qualquer sistema orientado a objetos. Há, entretanto, o lado sinistro dessa importante (e necessária) característica. Como o volume de comunicação e colaboração aumenta (isto é, à medida que o grau de “conexão” entre as classes aumenta), a complexidade do sistema também cresce. E à medida que a complexidade aumenta, a dificuldade de implementação, testes e manutenção do software também aumentam.

O **acoplamento** é uma medida qualitativa do grau com que as classes estão ligadas entre si. Conforme as classes (e os componentes) se tornam mais interdependentes, o acoplamento aumenta. Um objetivo importante no projeto de componentes é manter o acoplamento o mais baixo possível.

O acoplamento de classes pode se manifestar de uma série de formas. Lethbridge e Laganière [Let01] definem as seguintes categorias de acoplamento:

Acoplamento por conteúdo. Ocorre quando um componente “modifica de forma sub-reptícia os dados internos a um outro componente” [Let01]. Isso viola o encapsulamento — um conceito de projeto básico.

Acoplamento comum. Ocorre quando uma série de componentes faz uso de uma variável global. Embora algumas vezes isso seja necessário (por exemplo, para estabelecer valores-padrão aplicáveis em uma aplicação), o acoplamento comum pode levar à propagação de erros incontrolada e efeitos colaterais não previstos quando forem feitas modificações.

Acoplamento por controle. Ocorre quando a operação *A()* chama a operação *B()* e passa um flag de controle para *B*. O flag de controle “dirige” então a lógica de fluxo no interior de *B*. O problema com essa forma de acoplamento é que uma mudança não relacionada em *B* pode resultar na necessidade de alterar o significado do flag de controle que *A* passa. Se isso for menosprezado, acontecerá um erro.

Acoplamento “carimbo”. Ocorre quando **ClasseB** é declarada como um tipo para um argumento de uma operação da **ClasseA**. Como agora **ClasseB** faz parte da definição da **ClasseA**, modificar o sistema torna-se mais complexo.

Acoplamento por dados. Ocorre quando operações passam longas strings como argumentos de dados. A “largura de banda” da comunicação entre classes e componentes aumenta e a complexidade da interface cresce. Os testes e a manutenção são mais difíceis.

Acoplamento por chamadas de rotinas. Ocorre quando uma operação chama outra. Esse nível de acoplamento é comum e, quase sempre, necessário. Entretanto, realmente aumenta a conectividade de um sistema.

Acoplamento por uso de tipos. Ocorre quando o componente **A** usa um tipo de dados definido em um componente **B** (por exemplo, isso ocorre toda vez que “uma classe declarar uma variável de instância ou uma variável local como tendo outra classe para seu tipo” [Let01]). Se a definição de tipo mudar, todo componente que usa a definição também tem de ser alterado.

Acoplamento por inclusão ou importação. Ocorre quando um componente **A** importa ou inclui um pacote ou o conteúdo do componente **B**.

Acoplamento externo. Ocorre quando um componente se comunica ou colabora com componentes de infraestrutura (por exemplo, funções do sistema operacional, capacidade de bancos de dados, funções de telecomunicação). Embora esse tipo de acoplamento seja necessário, deve se limitar a um pequeno número de componentes ou classes em um sistema.

CASA SEGURA



Acoplamento em ação

Cena: Sala de Shakira.

Atores: Vinod e Shakira — membros da equipe de engenharia de software do CasaSegura que estão trabalhando na função de segurança.

Conversa:

Shakira: Pensei que tive uma grande ideia... Depois refleti um pouco mais a respeito e me pareceu não ser uma ideia tão boa assim. Por fim, a rejeitei, mas pensei que deveria apresentá-la a você.

Vinod: Certamente. Qual é a ideia?

Shakira: Bem, cada um dos sensores reconhece uma condição de alarme de algum tipo, certo?

Vinod (sorrindo): É por isso que os chamamos de sensores, Shakira.

Shakira (exasperada): Sarcasmo, Vinod, você tem que trabalhar suas habilidades interpessoais.

Vinod: Voltando ao que você dizia...

Shakira: Certo, de qualquer modo, imaginei... Por que não criar uma operação em cada objeto sensor chamada fazerCha-

mada() que colaboraria diretamente com o componente **ChamadaExterna**, enfim, com uma interface para o componente **ChamadaExterna**.

Vinod (pensativo): Você quer dizer que em vez de fazer com que a colaboração ocorra fora de um componente como **PainelControle** ou algo do gênero?

Shakira: Exato... Mas depois pensei que cada objeto sensor será associado ao componente **ChamadaExterna** e que isso significa que ele está indiretamente acoplado ao mundo exterior e... Bem, simplesmente imaginei que ele tornaria as coisas mais complicadas.

Vinod: Concordo. Nesse caso, é melhor deixar que a interface de sensores passe informações para o **PainelControle** e deixe que ele inicie a chamada telefônica. Além disso, diferentes sensores talvez resultem em diferentes números de telefone. Você não vai querer que o sensor armazene essas informações... Se elas mudarem...

Shakira: Parece que isso não está certo.

Vinod: A heurística de projeto para acoplamento nos diz que não é correto.

Shakira: Que seja...

Um software deve se comunicar interna e externamente. Consequentemente, acoplamento é uma realidade a ser enfrentada. Entretanto, o projetista deve se esforçar para reduzir o acoplamento sempre que possível e compreender as ramificações do acoplamento elevado quando não puder ser evitado.

10.3 CONDUÇÃO DE PROJETOS DE COMPONENTES

“Se eu tivesse mais tempo, teria escrito uma carta mais curta.”

Blaise Pascal



Se estiver trabalhando em um ambiente não orientado a objetos, as três primeiras etapas concentram-se no refinamento de objetos de dados e funções de processamento (transformações) identificadas como parte do modelo de requisitos.

No início deste capítulo citei que o projeto de componentes é de natureza elaborada. Temos de transformar informações de modelos de arquitetura e requisitos em uma representação de projeto que nos dê detalhes suficientes para orientar a atividade da construção (codificação e testes). As etapas a seguir representam um conjunto de tarefas típico para um projeto de componentes, quando ele é aplicado a um sistema orientado a objetos.

Etapla 1. Identificar todas as classes de projeto correspondentes ao domínio do problema. Usando o modelo de requisitos e de arquitetura, cada classe de análise e componente de arquitetura é elaborada conforme descrito na Seção 10.1.1.

Etapla 2. Identificar todas as classes de projeto correspondentes ao domínio de infraestrutura. Essas classes não são descritas no modelo de requisitos e normalmente não estão presentes no modelo de arquitetura; porém, têm de ser descritas nesse ponto. Conforme já dito, entre as classes e componentes dessa categoria temos os componentes de interfaces gráficas do usuário (muitas vezes disponíveis na forma de componentes reutilizáveis), em componentes de sistemas operacionais, bem como em componentes de administração de dados e objetos.

Etapla 3. Elaborar todas as classes de projeto que não são obtidas como componentes reutilizáveis. A elaboração requer que todas as interfaces, atributos e operações necessários para implementar a classe sejam descritos em detalhe. A heurística de projeto (por exemplo, coesão e acoplamento de componentes) deve ser considerada à medida que essa tarefa é conduzida.

Etapla 3a. Especificar detalhes de mensagens quando classes ou componentes colaboram entre si. O modelo de requisitos faz uso de um diagrama de colaboração para mostrar como as classes de análise colaboram entre si. À medida que o projeto de componentes prossegue, algumas vezes é útil mostrar os detalhes dessas colaborações especificando a estrutura das mensagens passadas entre objetos de um sistema. Embora essa atividade de projeto seja opcional, pode ser utilizada como um precursor da especificação de interfaces que mostram como os componentes em um sistema se comunicam e colaboram entre si.

A Figura 10.6 ilustra um diagrama de colaboração simples para o sistema de impressão discutido anteriormente. Três objetos, **ServiçoDeProdução**, **OrdemDeServiço** e **FilaDeServiço**, colaboram na preparação de um serviço de impressão a ser submetido ao fluxo de produção. São passadas mensagens entre os objetos conforme ilustrado pelas setas da figura. Durante a modelagem de requisitos as mensagens são especificadas conforme mostra a figura. Entretanto, à medida que o projeto prossegue, cada mensagem é elaborada por meio da expansão de sua sintaxe, da seguinte maneira [Ben02]:

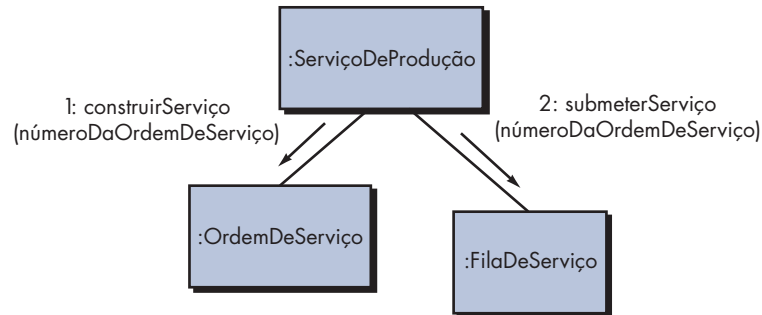
[condição de guarda] sequência de expressões (valor de retorno) :=
nome da mensagem (lista de argumentos)

em que uma [condição de guarda] é escrita em linguagem de restrição de objetos (*Object Constraint Language*, OCL)⁵ e que especifica qualquer conjunto de condições que devem ser atendidas antes de a mensagem poder ser enviada; **sequência de expressões** é um valor inteiro (ou outro indicador de ordem, por exemplo, 3.1.2) que indica a ordem sequencial em que uma mensagem é enviada; **(valor de retorno)** é o nome das informações retornadas por uma operação

⁵ A OCL é discutida brevemente no Apêndice 1.

FIGURA 10.6

Diagrama de colaboração com as mensagens



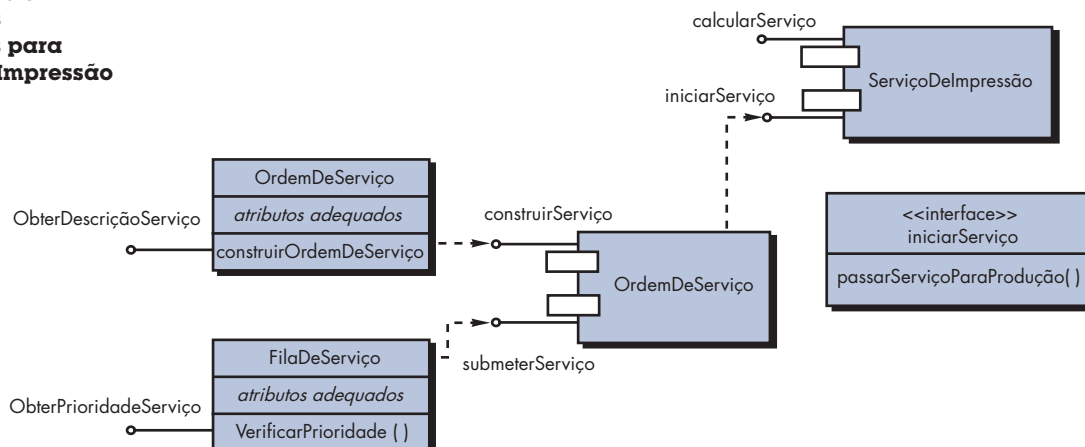
chamada pela mensagem; **nome da mensagem** identifica uma operação a ser chamada e (**lista de argumentos**) é a lista de atributos passados para a operação.

Etapa 3b. Identificar interfaces adequadas para cada componente. No contexto de projeto de componentes, uma interface UML é “um grupo de operações externamente visíveis (públicas). A interface não contém nenhuma estrutura interna, nenhum atributo, nenhuma associação...” [Ben02]. Colocado mais formalmente, interface equivale a uma classe abstrata que fornece uma conexão controlada entre as classes de projeto. A elaboração de interfaces é ilustrada na Figura 10.1. Em essência, operações definidas para o diagrama de classes são classificadas em uma ou mais classes abstratas. Todas as operações contidas em uma classe abstrata (a interface) devem ser coesas; elas devem apresentar processamento focado em uma função ou subfunção limitada.

Referindo-se à Figura 10.1, pode ser alegado que a interface *iniciarServiço* não apresenta coesão suficiente. Na realidade, ela realiza três subfunções distintas — criar uma ordem de serviço, verificar a prioridade do serviço e passar um serviço para produção. O projeto de interfaces deve ser refatorado. Uma abordagem poderia ser reexaminar as classes de projeto e definir uma nova classe **OrdemDeServiço** que cuidaria de todas as atividades associadas à montagem de uma ordem de serviço. A operação *construirOrdemDeServiço()* passa a fazer parte dessa classe. De modo similar, poderíamos definir uma classe **FilaDeServiço** que incorporaria a operação *verificarPrioridade()*. Uma classe **ServiçoDeProdução** englobaria todas as informações associadas a um serviço de produção a ser passado para o centro de produção. A interface *iniciarServiço* assumiria então a forma indicada na Figura 10.7. A interface *iniciarServiço* agora é coesa,

FIGURA 10.7

Interfaces de refatoração e definições de classes para ServiçoDeImpressão



focalizando uma função. As interfaces associadas a **ServiçoDeProdução**, **OrdemDeServiço** e **FilaDeServiço** são similarmente coesas.

Etapla 3c. Elaborar atributos e definir tipos de dados e estruturas de dados necessárias para implementá-los. Em geral, as estruturas e tipos de dados utilizados para definir atributos são definidas no contexto da linguagem de programação que será usada para implementação. A UML define o tipo de dados de um atributo usando a seguinte sintaxe:

nome : **tipo da expressão** = **valor inicial** {**string de propriedades**}

em que **nome** é o nome do atributo, **tipo da expressão** é o tipo de dados, **valor inicial** é o valor que o atributo assume quando um objeto é criado e **string de propriedades** define uma propriedade ou característica do atributo.

Durante a primeira iteração de projeto de componentes, os atributos normalmente são descritos por nomes. Referindo-se mais uma vez à Figura 10.1, a lista de atributos para **ServiçoDeImpressão** enumera apenas os nomes dos atributos. Entretanto, à medida que a elaboração do projeto prossegue, cada atributo é definido usando o formato de atributo UML citado. Por exemplo, **peso-TipoDoPapel** é definido da seguinte maneira:

peso-TipoDoPapel: **string** = "A" { contém 1 de 4 valores - A, B, C ou D }

que define **peso-TipoDoPapel** como uma variável de **string** inicializada para o valor A que pode assumir qualquer um dos quatro valores do conjunto {A,B,C, D}.

Se um atributo aparecer repetidamente ao longo de uma série de classes de projeto e tiver uma estrutura relativamente complexa, é melhor criar uma classe separada para acomodá-lo.

Etapla 3d. Descrever detalhadamente o fluxo de processamento contido em cada operação. Isso poderia ser concretizado usando-se um pseudocódigo baseado em linguagem de programação ou através de um diagrama de atividades UML. Cada componente de software é elaborado por meio de uma série de iterações que aplicam o conceito de refinamento gradual (Capítulo 8).

A primeira iteração define cada operação como parte da classe de projeto. Em cada caso, a operação deve ser caracterizada de modo a garantir alta coesão; a operação deve realizar uma única função ou subfunção determinada. A iteração seguinte faz pouco mais que expandir o nome da operação. Por exemplo, a operação *calcularCustoPapel()* indicada na Figura 10.1 pode ser expandida da seguinte maneira:

calcularCustoPapel (**peso, tamanho, cor**): **numérico**

Isso indica que *calcularCustoPapel ()* requer os atributos **peso**, **tamanho** e **cor** como entrada e retorna um valor numérico (na verdade um valor monetário) como saída.

Se o algoritmo necessário para implementar *calcularCustoPapel ()* for simples e largamente compreendido, talvez não seja necessário maior elaboração de projeto. O engenheiro de software que realiza a codificação fornece os detalhes necessários para implementar a operação. Entretanto, se o algoritmo for mais complexo ou enigmático, será necessária maior elaboração de projeto nesse estágio. A Figura 10.8 representa um diagrama de atividades UML para *calcularCustoPapel ()*. Quando os diagramas de atividades são usados para a especificação de projeto de componentes, em geral são representados em um nível de abstração ligeiramente maior que o do código-fonte. Uma abordagem alternativa — o uso de pseudocódigo para especificação de projeto — é discutida na Seção 10.5.3.

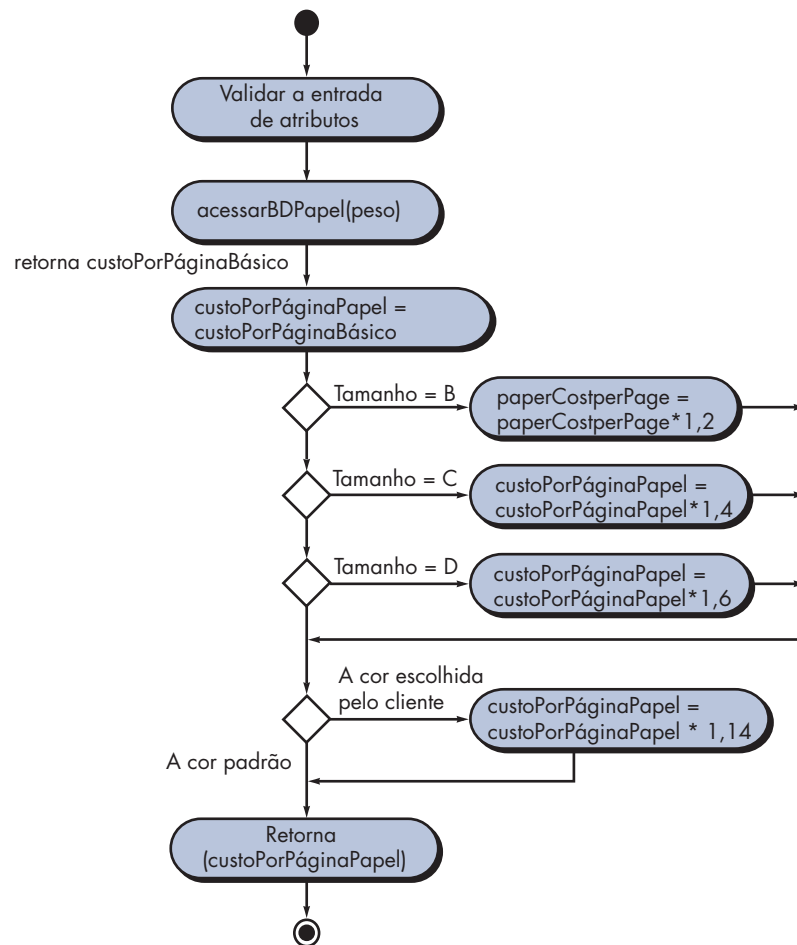
Etapla 4. Descrever fontes de dados persistentes (bancos de dados e arquivos) e identificar as classes necessárias para gerenciá-los. Os bancos de dados e arquivos normalmente transcendem a descrição de projeto de um componente individual. Na maioria dos casos, esses repositórios de dados persistentes são especificados inicialmente como parte do projeto da arquitetura. Entretanto, à medida que a elaboração do projeto



Use elaboração gradual à medida que for refinando o projeto de componentes. Pergunte sempre "Existe uma maneira pela qual isso possa ser simplificado e ainda assim obter o mesmo resultado?"

FIGURA 10.8

Diagrama de atividades UML para calcularCustoPapel()



prossegue, é útil fornecer detalhes adicionais sobre a estrutura e a organização das fontes de dados persistentes.

Etapla 5. Desenvolver e elaborar representações comportamentais para uma classe ou componente.

Usaram-se diagramas de estados UML como parte do modelo de requisitos para representar o comportamento externamente observável do sistema e o comportamento mais localizado de classes de análise individuais. Durante o projeto de componentes, algumas vezes é necessário modelar o comportamento de uma classe de projeto.

O comportamento dinâmico de um objeto (uma instanciação de uma classe de projeto à medida que o programa é executado) é afetado por eventos externos a ele e pelo estado atual (modo de comportamento) do objeto. Para compreender o comportamento dinâmico de um objeto, você deve examinar todos os casos de uso relevantes para a classe de projeto ao longo de sua vida. Estes fornecem informações que o ajudam a delinear os eventos que afetam o objeto e os estados em que o objeto reside à medida que o tempo passa e os eventos ocorrem. As transições entre estados (dirigidos por eventos) são representadas usando-se um diagrama de estado UML [Ben02], conforme ilustrado na Figura 10.9.

A transição de um estado (representado por um retângulo com cantos arredondados) para outro ocorre como consequência de um evento que assume a seguinte forma:

nome-evento (lista-parâmetros) [condição-de-guarda] / expressão de ação

em que **nome-evento** identifica o evento, **lista-parâmetros** incorpora dados associados ao evento, **condição-de-guarda** é escrito em OCL e especifica uma condição que deve ser atendida antes de o

implantação são elaborados na forma de instância naquele momento. Isso significa que o(s) ambiente(s) de sistema operacional e de hardware específicos utilizados é(são) especificado(s) e a localização de pacotes de componentes nesse ambiente é indicada.

Etapa 7. Refatorar toda representação de projetos de componentes e sempre considerar alternativas. Ao longo deste livro, tenho enfatizado que projeto é um processo iterativo. O primeiro modelo no nível de componentes que criamos não será tão completo, consistente ou acurado quanto o da *enésima* iteração aplicada ao modelo. É essencial refatorar à medida que o trabalho de projeto é conduzido.

Além disso, não se deve ter uma visão restrita. Sempre há soluções alternativas de projeto, e os melhores projetistas consideram todas (ou quase todas) elas antes de se decidirem pelo modelo de projetos final. Desenvolva alternativas e considere cuidadosamente cada uma delas, usando os princípios e conceitos apresentados no Capítulo 8 e neste capítulo.

10.4 PROJETO DE COMPONENTES PARA WEBAPPS

A fronteira entre conteúdo e função normalmente fica obscura quando se consideram sistemas e aplicações baseadas na Web (WebApps). Consequentemente, é razoável perguntar: O que é um componente WebApp?

No contexto do presente capítulo, um componente de WebApp é (1) uma função coesa bem definida que manipula conteúdo ou fornece processamento computacional ou de dados para um usuário final ou (2) um pacote coeso de conteúdo e funcionalidade que forneça ao usuário final alguma capacidade exigida. Consequentemente, o projeto de componentes para WebApps em geral incorpora elementos de projeto de conteúdo e de projeto funcional.

10.4.1 Projeto de conteúdo para componentes

O projeto de conteúdo para componentes concentra-se nos objetos de conteúdo e na maneira como poderiam ser empacotados para apresentação a um usuário final de uma WebApp. Consideremos, por exemplo, um recurso de vigilância de vídeo baseado na Web contido em **CasaSeguraGarantida.com**. Entre os diversos recursos, o usuário pode escolher e controlar qualquer uma das câmeras representadas como parte da planta, requisitar imagens em miniaturas capturadas por vídeo de todas as câmeras e exibir vídeos *streaming* de qualquer uma das câmeras. Além disso, o usuário poderá controlar o deslocamento e a ampliação de uma câmera usando ícones de controle apropriados.

Podemos definir uma série de componentes de conteúdo potenciais para o recurso de vigilância de vídeo: (1) os objetos de conteúdo que representam a distribuição de espaço (a planta) com ícones adicionais indicando a posição de sensores e câmeras de vídeo, (2) o conjunto de imagens em miniaturas capturadas por vídeo (cada um deles um objeto de dados distinto) e (3) a janela de vídeo *streaming* para uma câmera específica. A cada um dos componentes pode ser atribuído um nome e estes podem ser manipulados como um único pacote.

Consideremos uma planta que represente quatro câmeras estrategicamente colocadas no interior de uma casa. Por meio da solicitação do usuário, é capturado um quadro de cada câmera e estes identificados como um objeto de conteúdo gerado dinamicamente, **CapturaDeVídeoN**, em que *N* identifica as câmeras 1 a 4. Um componente de conteúdo, chamado **ImagensEm-Miniatura**, combina todos os quatro objetos de conteúdo **CapturaDeVídeoN** e os exibe na página de vigilância de vídeo.

A formalidade do projeto de conteúdo para componentes deve ser ajustada às características da WebApp a ser construída. Em muitos casos, os objetos de conteúdo não precisam ser organizados como componentes e podem ser manipulados individualmente. Entretanto, à medida que o tamanho e a complexidade (da WebApp, dos objetos de conteúdo e suas inter-relações) forem crescendo, talvez seja necessário organizar o conteúdo para permitir uma manipulação de pro-

jeto e referência mais fácil.⁶ Além disso, se o conteúdo for altamente dinâmico (por exemplo, o conteúdo para um site de leilões on-line), é importante estabelecer um claro modelo estrutural que incorpore os componentes de conteúdo.

10.4.2 Projeto funcional para componentes

As aplicações Web modernas oferecem funções de processamento cada vez mais sofisticadas que (1) realizam processamento localizado para gerar capacidades de conteúdo e navegação de maneira dinâmica, (2) fornecem recursos de processamento de dados ou cálculos apropriados para a área de aplicação das WebApps, (3) fornecem acesso e consultas sofisticadas a bancos de dados ou (4) estabelecem interfaces de dados com sistemas corporativos externos. Para conseguirmos esses (e muitos outros) recursos, projetamos e construímos componentes funcionais para WebApps que sejam similares a componentes de software para software convencional.

A funcionalidade da WebApp é fornecida através de uma série de componentes desenvolvidos em paralelo com o projeto das informações para garantir que sejam consistentes. Em essência, partimos da consideração do modelo de requisitos, bem como da arquitetura das informações iniciais e, em seguida, examinamos como a funcionalidade afeta a interação do usuário com a aplicação, as informações que são apresentadas e as tarefas de usuário realizadas.

Durante o projeto da arquitetura, o conteúdo e a funcionalidade da WebApp são combinados para criar uma *arquitetura funcional*, que é uma representação do domínio funcional da WebApp e descreve os componentes funcionais fundamentais da WebApp e como interagem entre si.

Por exemplo, as funções de deslocamento e ampliação de imagem para o recurso de vigilância por vídeo do **CasaSeguraGarantida.com** são implementadas como parte de um componente **ControleDeCâmera**. Como alternativa, o deslocamento e a ampliação de imagem podem ser implementados como operações, *deslocar()* e *ampliar()*, que fazem parte da classe **Câmera**. Em ambos os casos, a funcionalidade implicada pelas funções de deslocamento e ampliação de imagem deve ser implementada como módulos contidos em **CasaSeguraGarantida.com**.

10.5 PROJETO DE COMPONENTES TRADICIONAIS

PONTO-CHAVE

Programação estruturada é uma técnica de projeto que restringe o fluxo lógico a três construções: sequência, condição e repetição.

Os fundamentos para o projeto de componentes para componentes de software tradicionais⁷ foram formados no início dos anos 1960 e foram solidificados com o trabalho de Edsger Dijkstra e seus colegas ([Boh66], [Dij65], [Dij76b]). No final dos anos 1960, Dijkstra e outros propuseram o uso de um conjunto de construções lógicas restritas a partir das quais qualquer programa poderia ser formado. As construções enfatizavam “a manutenção do domínio funcional”. Ou seja, cada construção possuía uma estrutura lógica previsível e entra-se nela pela parte superior e sai-se pela inferior, possibilitando a um leitor seguir mais facilmente o fluxo procedural.

As construções são sequência, condição e repetição. *Sequência* implementa etapas de processamento essenciais na especificação de qualquer algoritmo. *Condição* fornece a facilidade para processamento seletivo baseado em alguma ocorrência lógica e *repetição* possibilita o looping. Essas três construções são fundamentais para a *programação estruturada* — uma importante técnica para projetos de componentes.

As construções estruturadas foram propostas para limitar o projeto procedural de software a um pequeno número de estruturas lógicas previsíveis. As métricas de complexidade (Capítulo 23) indicam que o uso de construções estruturadas reduz a complexidade dos programas e, consequentemente, facilitam a legibilidade, a realização de testes e manutenção. O uso de um número

⁶ Os componentes de conteúdo também podem ser reutilizados em outras WebApps.

⁷ Um componente de software tradicional implementa um elemento de processamento que trata uma função ou subfunção no domínio do problema ou alguma capacidade no domínio da infraestrutura. Em geral denominados módulos, procedimentos ou sub-rotinas, os componentes tradicionais não encapsulam dados da mesma forma que os componentes orientados a objetos.

limitado de construções lógicas também contribui para o processo de compreensão humana chamado pelos psicólogos de *agrupamento*. Para compreender esse processo, considere a maneira como você está lendo esta página. Não lemos as letras individualmente, mas reconhecemos padrões ou grupos de letras que formam as palavras ou frases. As construções estruturadas são grupos lógicos que permitem a um leitor reconhecer elementos procedurais de um módulo, em vez de ter de ler o projeto ou código linha por linha. A compreensão aumenta quando são encontrados padrões lógicos reconhecíveis.

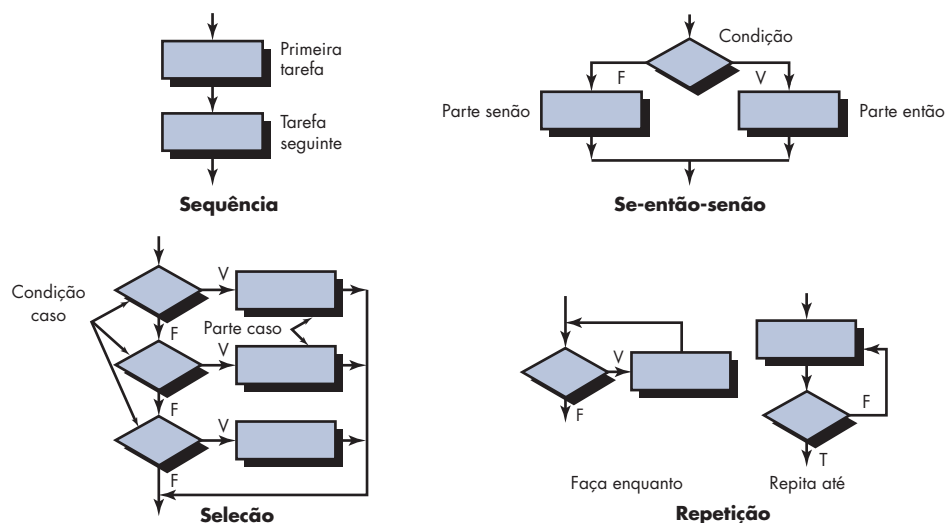
Qualquer programa, independentemente da área de aplicação ou de sua complexidade técnica, pode ser projetado e implementado usando-se apenas as três construções estruturadas. Entretanto, deve-se frisar que o uso dogmático apenas dessas construções pode, algumas vezes, provocar dificuldades práticas. A Seção 10.5.1 considera a questão de forma mais detalhada.

10.5.1 Notação gráfica em projeto

“Uma figura vale mais do que mil palavras”, porém é muito importante saber qual figura e quais mil palavras. Não há nenhuma dúvida de que ferramentas gráficas, como o fluxograma ou o diagrama de atividades UML, dispõem de padrões pictóricos muito úteis que prontamente representam detalhes procedurais. Entretanto, se as ferramentas gráficas forem mal utilizadas, a figura errada poderá levar a um software errado.

O diagrama de atividades permite que representemos sequência, condição e repetição — todos elementos da programação estruturada — e é um descendente de uma representação pictórica de projetos anterior (ainda largamente utilizada) chamada *fluxograma*, que assim como o diagrama de atividades, é pictoricamente bastante simples. É usado um retângulo para indicar uma etapa de processamento. Um losango representa uma condição lógica e setas mostram o fluxo de controle. A Figura 10.10 ilustra três construções estruturadas. A *sequência* é representada na forma de dois retângulos de processamento conectados por meio de uma linha (seta) de controle. Uma *condição*, também chamada *se-então-senão*, é representada por um losango de decisão que, se verdadeira, faz com que o processamento da *parte então* ocorra e, se falsa, chama o processamento da *parte senão*. Uma *repetição* é representada usando-se duas formas ligeiramente diferentes. O *faça enquanto* testa uma condição e executa uma tarefa em loop repetitivamente até que a condição seja verdadeira. Um *repita até* primeiro executa a tarefa em loop e depois testa a condição e repete a tarefa até que a condição falhe. A construção de *seleção*

FIGURA 10.10
Construções de fluxograma



(ou *seleção de caso*) mostrada na figura é, na verdade, uma extensão de *se-então-senão*. Um parâmetro é testado por decisões sucessivas até a ocorrência de uma condição verdadeira e um caminho de processamento de um caso é executado.

Em geral, o uso dogmático apenas das construções estruturadas pode introduzir ineficiência quando for necessária uma saída de um conjunto de laços aninhados ou de condições aninhadas. Mais importante ainda, a complicação adicional de todos os testes lógicos ao longo do caminho de saída pode confundir o fluxo de controle do software, aumentar a possibilidade de erros e ter um impacto negativo na legibilidade e na facilidade de manutenção. O que fazer então?

Restam duas opções: (1) a representação procedural ser redesenhada de modo que a “ramificação de saída” não seja necessária em um ponto aninhado no fluxo de controle ou (2) as construções estruturadas serem violadas de uma maneira controlada; ou seja, é criada uma ramificação com restrição para fora do fluxo aninhado. A opção 1 é, obviamente, a abordagem ideal, porém a opção 2 pode ser aceita sem violar o espírito da programação estruturada.

10.5.2 Notação tabular de projeto



Use uma tabela de decisão quando for encontrado um conjunto complexo de condições e ações em um componente.

Em diversas aplicações de software, pode ser necessário um módulo para avaliar uma combinação complexa de condições e selecionar ações apropriadas com base nessas condições. As *tabelas de decisão* [Hur83] dispõem de uma notação que transforma ações e condições (descritas em uma narrativa de processamento ou caso de uso) em forma tabular. A tabela é difícil de ser mal interpretada e até poderia ser usada como entrada legível por máquina para um algoritmo dirigido por tabelas.

A organização de uma tabela de decisão é ilustrada na Figura 10.11. Na figura, a tabela é dividida em quatro seções. O quadrante superior esquerdo contém uma lista de todas as condições. O quadrante inferior esquerdo contém uma lista de todas as ações possíveis com base em combinações de condições. Os quadrantes à direita formam uma matriz que indica combinações de condição e as ações correspondentes que ocorrerão a determinada combinação. Consequentemente, cada coluna da matriz pode ser interpretada como uma *regra de processamento*. As etapas a seguir são aplicadas para desenvolver uma tabela de decisão:



Como construir uma tabela de decisões?

1. Listar todas as ações que podem ser associadas a um procedimento (ou componente) específico.
2. Listar todas as condições (ou decisões tomadas) durante a execução do procedimento.
3. Associar conjuntos de condições específicos a ações específicas, eliminando combinações de condições impossíveis; como alternativa, desenvolver toda permutação de condições possível.
4. Definir regras por meio da indicação de quais ações ocorrem para um conjunto de condições.

Para ilustrarmos o emprego de uma tabela de decisão, consideremos o seguinte exemplo de um caso de uso informal que acaba de ser proposto para o sistema de gráfica:

São definidos três tipos de clientes: um cliente comum, um cliente prata e um cliente ouro (esses tipos são atribuídos conforme o volume de negócios realizado pelo cliente com a gráfica nos últimos 12 meses). Um cliente comum terá preços e prazos de entrega normais. Um cliente prata obtém um desconto de 8% em todos os orçamentos e é colocado à frente de todos os clientes comuns na fila de serviços. Um cliente ouro tem uma redução de 15% nos preços cotados e é colocado à frente de todos os clientes comuns e prata na fila de serviços. Um desconto especial de x por cento, além de outros descontos, poderá ser aplicado aos orçamentos de qualquer cliente a critério da gerência.

A Figura 10.11 ilustra a representação de uma tabela de decisão do caso de uso informal anterior. Cada uma das seis regras indica uma das seis condições viáveis. Como regra geral, a

FIGURA 10.11

**Nomenclatura de
Tabelas de decisão**

Condições	Regras					
	1	2	3	4	5	6
Cliente comum	V	V				
Cliente prata			V	V		
Cliente ouro					V	V
Desconto especial	F	V	F	V	F	V
Ações						
Nenhum desconto	✓					
Aplicar desconto de 8%			✓	✓		
Aplicar desconto de 15%					✓	✓
Aplicar desconto adicional de x%		✓		✓		✓

tabela de decisão pode ser usada efetivamente para complementar outras notações procedurais de projeto.

10.5.3 Linguagem de projeto de programas

A *linguagem de projeto de programas* (*Program Design Language*, PDL), também chamada *inglês estruturado* ou *pseudocódigo*, incorpora a estrutura lógica de uma linguagem de programação com a habilidade expressiva de forma livre de uma linguagem natural (o inglês, por exemplo). Texto narrativo (em inglês, por exemplo) é incorporado à sintaxe de programação parecida com uma língua. Ferramentas automatizadas (por exemplo, [Cai03]) podem ser utilizadas para melhorar a aplicação da PDL.

Uma sintaxe PDL básica deve incluir construções para a definição de componentes, descrição de interfaces, declaração de dados, estruturação de blocos, construções de condição, de repetição e de entrada/saída (I/O). Deve-se notar que a PDL pode ser estendida para incluir palavras-chave para multitarefas e/ou processamento concorrente, tratamento de interrupções, sincronização entre processos e muitas outras características. O projeto de aplicações para a qual a PDL será usada deve ditar a forma final para a linguagem de projeto. O formato e a semântica para algumas dessas construções PDL são apresentados no exemplo a seguir.

Para ilustrarmos o uso da PDL, consideremos um projeto procedural para a função de segurança do *CasaSegura* discutida em capítulos anteriores. O sistema monitora alarmes para incêndio, fumaça, ladrões, água e temperatura (por exemplo, o sistema de aquecimento falha enquanto o proprietário do imóvel encontra-se fora durante o inverno) e produz um alarme sonoro e chama o serviço de monitoramento, gerando uma mensagem de voz sintetizada.

Recorde-se que a PDL *não* é uma linguagem de programação. Pode-se adaptar conforme a necessidade sem a preocupação em relação a erros de sintaxe. Entretanto, o projeto para software de monitoramento teria de ser revisado (você vê algum problema nisso?) e mais refinado antes que o código pudesse ser escrito. As PDLs⁸ a seguir fornecem uma elaboração do projeto procedural para uma versão inicial de um componente de gerenciamento de alarme.

⁸ O nível de detalhe representado pela PDL é definido localmente. Algumas pessoas preferem uma descrição orientada a uma linguagem mais natural, enquanto outros preferem algo mais próximo de código.


```

componente gerenciamentoAlarme;
O intuito deste componente é o de gerenciar chaves de um painel de controle e a entrada dos sensores por
tipo e atuar sob qualquer condição de alarme que seja encontrada.
  fixar valores default para
    estadosDoSistema (valor de retorno), todos os itens de dados
  iniciar todas as portas do sistema e reiniciar todo o hardware
  verificar chavesDoPainelDeControle (cpc)
    se cpc = "teste" então iniciar o alarme configurado em "ligado"
    se cpc = "alarmeDesligado" então iniciar o alarme configurado em "desligado"
    se cpc = "novoValorDeLimite" então iniciar o teclado
    se cpc = "alarmeDeAssaltoDesligado" então desativarAlarme
    .
    .
    .
  default para cpc = nada
  reiniciar todos os valoresDeSinais e chaves
  faça para todos os sensores
    chamar procedimento verificarSensor retornando valorDoSinal
    se valorDoSinal > limite[tipoDeAlarme]
      então mensagemTelefonica = mensagem [tipoAlarme]
      configurar campainhaDoAlarme para "ligada" para tempoAlarmeEmSegundos
      configurar estadoDoSistema = "condiçãoAlarme"
      par-início
        chamar procedimento alarme "ligado", tempoAlarmeEmSegundos
        chamar procedimento telefônico configurado para tipoAlarme, númeroDoTelefone
      par-fim
    senão saia
  fim-se
fim faça-para
fim gerenciamentoAlarme

```

Note que o projetista do componente **gerenciamentoAlarme** usou a construção **par-início... par-fim** que especifica um bloco paralelo. Todas as tarefas especificadas no bloco **par-início** são executadas em paralelo. Nesse caso, detalhes de implementação não são considerados.

10.6 DESENVOLVIMENTO BASEADO EM COMPONENTES

No contexto da engenharia de software, a reutilização é uma ideia ao mesmo tempo antiga e nova. Os programadores têm reutilizado ideais, abstrações e processos desde os primórdios da computação, mas a abordagem inicial para a reutilização era *ad hoc*. Hoje em dia, sistemas computacionais complexos e de alta qualidade devem ser construídos em prazos muito curtos e exigem uma abordagem mais organizada para a reutilização.

A engenharia de software baseada em componentes (CBSE) é um processo que enfatiza o projeto e a construção de sistemas baseados em computadores usando “componentes” de software reutilizáveis. Clements [Cle95] descreve a CBSE da seguinte maneira:

A [CBSE] incorpora a filosofia “compre, não construa” defendida por Fred Brooks e outros. Da mesma forma que sub-rotinas iniciais liberavam o programador de pensar em detalhes, a [CBSE] transfere a ênfase de programar software para compor sistemas de software. A implementação deu lugar à integração como foco.

Mas surge uma série de questões. É possível construir sistemas complexos montando-os por meio de um catálogo de componentes de software reutilizáveis? Isso pode ser realizado de maneira efetiva em termos de custo e tempo? Podem ser estabelecidos incentivos apropriados para

“Engenharia de domínio significa encontrar características comuns entre os sistemas para identificar componentes que possam ser aplicados a vários sistemas...”

Paul Clements



O processo de análise discutido nesta seção concentra-se em componentes reutilizáveis. Entretanto, a análise de sistemas comerciais de prateleira completos (por exemplo, aplicações de comércio eletrônico, aplicações para automação da equipe de vendas) também podem fazer parte da análise de domínio.

encorajar os engenheiros de software para a reutilização em vez de reinventar? A gerência está disposta a incorrer na despesa adicional associada à criação de componentes de software reutilizáveis? A biblioteca de componentes necessários para realizar a reutilização pode ser criada para torná-la acessível aos que precisam dela? Os componentes que realmente existem podem ser encontrados por aqueles que precisam deles?

Cada vez mais, a resposta a cada uma dessas questões é “sim”. No restante desta seção, examinaremos algumas questões que devem ser consideradas para tornar a CBSE bem-sucedida em uma organização de engenharia de software.

10.6.1 Engenharia de domínio

O objetivo da *engenharia de domínio* é identificar, construir, catalogar e disseminar um conjunto de componentes de software que tenham aplicabilidade em software existente e futuro em determinado domínio de aplicação.⁹ O objetivo geral é estabelecer mecanismos que permitam aos engenheiros de software compartilhar esses componentes — para reutilizá-los — durante o trabalho em sistemas novos e existentes. A engenharia de *domínio* abrange três atividades principais — análise, construção e disseminação.

A abordagem geral para a *análise de domínio* em geral é caracterizada no contexto da engenharia de software orientada a objetos. As etapas no processo são definidas como:

1. Definir o domínio a ser investigado.
2. Classificar os itens extraídos do domínio.
3. Coletar uma amostra representativa das aplicações do domínio.
4. Analisar cada aplicação na amostra e definir classes de análise.
5. Desenvolver um modelo de requisitos para as classes.

É importante notar que a análise de domínio aplica-se a qualquer paradigma de engenharia de software e poderia ser aplicada tanto em desenvolvimento convencional quanto orientado a objetos.

10.6.2 Qualificação, adaptação e composição de componentes

A engenharia de *domínio* fornece a biblioteca de componentes reutilizáveis necessários para a engenharia de software baseada em componentes. Alguns desses componentes reutilizáveis são desenvolvidos internamente, outros podem ser extraídos de aplicações existentes e outros ainda podem ser adquiridos de terceiros.

Infelizmente, a existência de componentes reutilizáveis não garante que possam ser integrados de forma fácil e efetiva à arquitetura escolhida para uma nova aplicação. É por essa razão que uma sequência de atividades de desenvolvimento baseado em componentes é aplicada quando um componente é proposto para uso.

Qualificação de componentes. A qualificação de componentes garante que um componente candidato irá realizar a função necessária, se ele “se encaixará” adequadamente ao estilo de arquitetura (Capítulo 9) especificada para o sistema e apresentará as características de qualidade (por exemplo, desempenho, confiabilidade, usabilidade) exigidas para a aplicação.

Uma descrição da interface fornece informações úteis sobre a operação e o uso de um componente de software, porém não fornece todas as informações necessárias para determinar se um componente proposto pode, de fato, ser reutilizado efetivamente em nova aplicação. Entre os muitos fatores considerados durante a qualificação de componentes temos [Bro96]:

- Interface de programas aplicativos (*application programming interface*, API).
- Ferramentas de desenvolvimento e integração exigidas pelo componente.

Quais fatores são considerados durante a qualificação de componentes?

⁹ No Capítulo 9 referimo-nos a gêneros de arquitetura que identificam domínios de aplicação específicos.

- Requisitos de tempo de execução, incluindo o emprego de recursos (por exemplo, memória ou armazenamento), timing ou velocidade e protocolo de redes.
- Requisitos de serviços, incluindo interfaces para sistemas operacionais e suporte de outros componentes.
- Características de segurança, incluindo controles de acesso e protocolo de autenticação.
- Pressupostos implícitos de projetos, incluindo o uso de algoritmos numéricos e não numéricos.
- Tratamento de exceções.

Cada um dos fatores é relativamente fácil de avaliar quando são propostos componentes reutilizáveis desenvolvidos internamente. Se forem aplicadas práticas adequadas de engenharia de software durante o desenvolvimento de um componente, poderão ser dadas respostas àquelas questões implícitas na lista. Entretanto, é muito mais difícil determinar o funcionamento interno de componentes comerciais de prateleira (*commercial off-the-shelf*, COTS) ou de terceiros, pois a única informação disponível pode ser a especificação da própria interface.

Adaptação de componentes. Em condições ideais, a engenharia de domínio cria uma biblioteca de componentes que pode ser facilmente integrada à arquitetura de uma aplicação. Para termos a “facilidade de integração” subentende-se que: (1) métodos consistentes de gerenciamento de recursos tenham sido implementados para todos os componentes da biblioteca, (2) atividades comuns como gerenciamento de dados existam para todos os componentes e (3) as interfaces internas da arquitetura e com ambiente externo tenham sido implementadas de maneira consistente.

Na realidade, mesmo após qualificar um componente para uso na arquitetura de uma aplicação, podem ocorrer conflitos em uma ou mais das áreas apenas citadas. Para evitá-los, algumas vezes é usada uma técnica de adaptação chamada *empacotamento de componentes* [Bro96]. Quando uma equipe de software tem acesso completo ao projeto e código internos de um componente (o que normalmente não ocorre a menos que sejam usados componentes comerciais de prateleira com código-fonte aberto), é aplicado um *empacotamento caixa branca*. Da mesma forma que seu equivalente em testes de software (Capítulo 18), o empacotamento caixa branca examina os detalhes de processamento interno do componente e realiza modificações no código para eliminar qualquer conflito. O *empacotamento caixa cinza* é aplicado quando a biblioteca de componentes oferece uma linguagem de extensão de componentes ou uma API que possibilitem a eliminação ou o mascaramento de conflitos. O *empacotamento caixa preta* requer a introdução de pré e pós-processamento na interface de componentes para eliminar ou mascarar conflitos. Temos de determinar se o esforço exigido para empacotar um componente adequadamente se justifica ou se vale mais criar um componente personalizado (projetado para eliminar os conflitos encontrados).

Composição de componentes. A tarefa de composição de componentes monta componentes qualificados, adaptados e construídos para compor a arquitetura estabelecida a uma aplicação. Para tal, deve ser estabelecida uma infraestrutura para vincular os componentes a um sistema operacional. A infraestrutura (em geral uma biblioteca de componentes especializados) fornece um modelo para a coordenação de componentes e serviços específicos que habilitam os componentes a se coordenarem entre si e realizarem tarefas comuns.

Devido ao potencial impacto da reutilização e CBSE no setor de software ser enorme, várias companhias importantes e consórcios da indústria propuseram padrões para software de componentes.¹⁰

OMG/CORBA. O Object Management Group (OMG) publicou uma *arquitetura de agentes de solicitação de objetos comuns* (*common object request broker architecture*, CORBA). Um *agente de solicitação de objetos* (*object request broker*, ORB) oferece uma série de serviços



Além de avaliarmos se o custo da adaptação para a reutilização se justifica, também devemos avaliar se a obtenção da funcionalidade e desempenho exigidos, pode ser feita de modo efetivo em termos de custo.

WebRef

As informações mais recentes sobre CORBA podem ser obtidas em www.omg.org.

¹⁰ Greg Olsen [Ols06] apresenta uma excelente discussão sobre empreendimentos do setor no passado e no presente para tornar a CBSE uma realidade.

que possibilita que componentes (objetos) reutilizáveis se comuniquem com outros componentes, independentemente de sua localização em um sistema.

WebRef

As informações mais recentes sobre JavaBeans podem ser obtidas em java.sun.com/produtos/javabeans/docs/.

Microsoft COM e .NET. A Microsoft desenvolveu um *modelo de objetos de componentes* (*component object model*, COM) que fornece uma especificação para uso de componentes produzidos por vários fabricantes em uma única aplicação que rode no sistema operacional Windows. Do ponto de vista da aplicação, “o foco não é em como [os objetos COM são] implementados, mas sim apenas no fato de que o objeto possui uma interface que ele registra no sistema e que usa o sistema de componentes para se comunicar com outros objetos COM” [Har98a]. A estrutura Microsoft .NET engloba COM e fornece uma biblioteca de classes reutilizáveis que abrange um largo espectro de domínios de aplicação.

WebRef

As informações mais recentes sobre COM e .NET podem ser obtidas em www.microsoft.com/COM e msdn2.microsoft.com/en-us/netframework/default.aspx.

Componentes JavaBeans da Sun. O sistema de componentes JavaBeans é uma infraestrutura CBSE independente de plataforma e portátil desenvolvida com o emprego da linguagem de programação Java. O sistema de componentes JavaBeans engloba um conjunto de ferramentas, denominado BDK (*Bean Development Kit*), que permite aos desenvolvedores: (1) analisar como os Beans (componentes) existentes funcionam, (2) personalizar seus comportamentos e aparências, (3) estabelecer mecanismos para coordenação e comunicação, (4) desenvolver Beans personalizados para uso em uma aplicação específica e (5) testar e avaliar o comportamento do Bean.

Nenhum desses padrões domina o mercado. Embora muitos desenvolvedores tenham adotado como algum deles, é provável que grandes empresas de software optem por adotar um padrão baseado nas plataformas e categorias de aplicação escolhidas.

10.6.3 Análise e projeto para reutilização

Embora o processo de CBSE encoraje o uso de componentes de software existentes, há situações em que componentes de software novos têm de ser desenvolvidos e integrados ao software de prateleira e componentes desenvolvidos internamente já existentes. Pelo fato de novos componentes tornarem-se membros da biblioteca de componentes reutilizáveis desenvolvida dentro da empresa, eles devem ser projetados tendo em vista a reutilização.

Conceitos de projeto como abstração, encapsulamento, independência funcional, refinamento e programação estruturada, juntamente com métodos, testes e garantia da qualidade de software (*software quality assurance*, SQA) orientados a objetos, bem como métodos para verificação da correção (Capítulo 21), todos eles contribuem para a criação de componentes de software reutilizáveis. Nesta subseção, consideraremos questões específicas à reutilização complementares às sólidas práticas de engenharia de software.

O modelo de requisitos é analisado para determinar aqueles elementos que apontam para componentes reutilizáveis existentes. Elementos do modelo de requisitos são comparados com as descrições de componentes reutilizáveis em um processo algumas vezes conhecido como “correspondência da especificação” [Bel95]. Se a correspondência da especificação apontar para um componente existente que atenda às necessidades da aplicação atual, poderemos extrair o componente de uma biblioteca (repositório) de reúso e usá-lo no projeto de um novo sistema. Se os componentes não puderem ser encontrados (se não existir nenhuma correspondência), é criado um novo componente. É nesse ponto — quando se inicia a criação de um novo componente — que o *projeto visando a reutilização* (*design for reuse*, DFR) deve ser considerado.

Conforme já dito, o DFR requer a aplicação de conceitos e princípios de projeto de software sólidos (Capítulo 8). Porém as características do domínio de aplicação também devem ser consideradas. Binder [Bin93] sugere uma série de questões-chave¹¹ que formam a base para o projeto visando a reutilização:

¹¹ Em geral, as preparações DFR devem ser realizadas como parte da engenharia de domínios.



O DFR pode ser bem difícil quando componentes tiverem de ser interfaceados ou integrados com sistemas legados ou com vários sistemas cuja arquitetura e protocolos de interfaceamento são inconsistentes.

Dados padronizados. O domínio de aplicação deve ser investigado e estruturas de dados globais padronizadas (por exemplo, estruturas de arquivos ou um banco de dados completo) devem ser identificadas. Todos os componentes de projeto podem então ser caracterizados para fazer uso dessas estruturas de dados padronizadas.

Protocolos de interface padronizados. Devem ser estabelecidos três níveis de protocolo de interface: a natureza das interfaces intramodulares, o projeto de interfaces técnicas externas (não humanas) e a interface homem-máquina.

Templates de programas. É escolhido um estilo de arquitetura (Capítulo 9) e que pode servir como um template para o projeto da arquitetura do novo software.

Uma vez estabelecidos os templates de programas, de interfaces e de dados, temos uma estrutura na qual criar o projeto. Componentes novos que se adaptem a essa estrutura têm maior probabilidade de posterior reutilização.

10.6.4 Classificação e recuperação de componentes

Consideremos uma grande biblioteca de universidade. Centenas de milhares de livros, periódicos e outras fontes de informação estão disponíveis para uso. Porém, é preciso desenvolver um esquema de classificação e acesso a esses recursos. Para navegar pelo enorme volume de informações, os bibliotecários definiram um esquema de classificação que inclui o código de classificação de Dados Internacionais de Catalogação na Publicação, palavras-chave, nomes de autores e outras entradas de índice. Todos os parâmetros possibilitam ao usuário encontrar um recurso necessário de forma rápida e fácil.

Consideremos agora um grande repositório de componentes. Dezenas de milhares de componentes de software reutilizáveis residem nele. Mas como encontrar aquele que precisamos? Para responder a essa pergunta, surge uma segunda: Como descrever componentes de software em termos inequívocos e classificáveis? Essas são perguntas difíceis e ainda não se chegou a uma resposta definitiva. Na presente seção exploraremos os rumos atuais que permitirão a futuros engenheiros de software navegarem em bibliotecas de reutilização.

Um componente de software reutilizável pode ser descrito de várias maneiras, porém, uma descrição ideal engloba aquilo que Tracz [Tra95] denominou *modelo 3C* — conceito, conteúdo e contexto. O *conceito* de um componente de software é “a descrição daquilo que o componente faz” [Whi95]. A interface para o componente é totalmente descrita e a semântica — representada no contexto das pré e pós-condições — é identificada. O conceito deve comunicar o objetivo do componente. O *conteúdo* de um componente descreve como o conceito é realizado. Em essência, o conteúdo são informações que ficam ocultas para usuários eventuais e que precisam ser conhecidas apenas por aqueles que pretendem modificar ou testar o componente. O *contexto* posiciona um componente de software reutilizável em seu domínio de aplicabilidade. Por meio da especificação de características conceituais, operacionais e de implementação, o contexto possibilita a um engenheiro de software encontrar o componente apropriado para atender os requisitos da aplicação.

Para serem úteis em um ambiente prático, conceito, conteúdo e contexto devem ser traduzidos em um esquema de especificação concreto. Dezenas de trabalhos e artigos foram escritos tratando os esquemas de classificação para componentes de software reutilizáveis (veja, por exemplo, [Cec06] para uma visão geral das tendências atuais).

A classificação nos permite encontrar e recuperar possíveis componentes reutilizáveis, porém o ambiente de reutilização deve existir para integrar efetivamente tais componentes. Um ambiente de reutilização apresenta as seguintes características:

- Um banco de dados de componentes capaz de armazenar componentes de software e as informações de classificação necessárias para recuperá-las.
- Um sistema de gerenciamento de bibliotecas que dê acesso ao banco de dados.

Quais são as principais características de um ambiente de reutilização de componentes?

- Um sistema de recuperação de componentes de software (por exemplo, um agente de solicitação de objetos) que permita a uma aplicação-cliente recuperar componentes e serviços do servidor de bibliotecas.
- As ferramentas CBSE que oferecem suporte à integração de componentes reutilizados em um novo projeto ou implementação.

Cada uma das funções interage com ou está embutida em uma biblioteca de reutilização.

A *biblioteca de reutilização* é um elemento de um repositório de software maior (Capítulo 22) e oferece recursos para o armazenamento de componentes de software e uma ampla gama de artefatos reutilizáveis (por exemplo, especificações, projetos, padrões, estruturas, trechos de código, casos de teste, guias de usuário). A biblioteca abrange um banco de dados e as ferramentas necessárias para consultar o banco de dados e recuperar componentes dele. O esquema de classificação dos componentes serve como base para consultas à biblioteca.

As consultas são em geral caracterizadas usando-se o elemento contexto do modelo 3C descrito anteriormente nesta seção. Se uma consulta inicial resultar em uma lista volumosa de possíveis componentes, a consulta deve ser refinada para restringir a lista. As informações de conceito e conteúdo são então extraídas (depois de possíveis componentes terem sido encontrados) para ajudar na seleção do componente apropriado.

WebRef

Um conjunto completo de recursos sobre CBSE pode ser encontrado em www.cbd-hq.com/.



CBSE

Objetivo: Apoiar a modelagem, projeto, revisão e integração de componentes de software como parte de um sistema mais amplo.

Mecânica: A mecânica das ferramentas varia. Em geral, as ferramentas CBSE auxiliam em uma ou mais das seguintes capacidades: especificação e modelagem da arquitetura de software, navegação e seleção de componentes de software disponíveis; integração de componentes.

Ferramentas representativas¹²

ComponentSource (www.componentsource.com) oferece uma ampla gama de componentes (e ferramentas) de software comerciais suportada em diversos padrões de componentes.

FERRAMENTAS DO SOFTWARE

Component Manager, desenvolvida pela Flashline (www.flashline.com), “é uma aplicação que permite, promove e mede a reutilização de componentes de software”.

Select Component Factory, desenvolvida pela Select Business Solutions (www.selectbs.com), “é um conjunto integrado de produtos para projeto de software, revisão de software,

gerenciamento de serviços/componentes, gerenciamento de requisitos e geração de código”.

Software Through Pictures-ACD, distribuída pela Aonix (www.aonix.com), permite modelagem completa empregando a UML para a arquitetura dirigida a modelos OMG — uma abordagem aberta e desvinculada de qualquer fornecedor específico para CBSE.

10.7 RESUMO

O processo para projeto de componentes abrange uma sequência de atividades que reduz lentamente o nível de abstração com o qual um software é representado. Em última instância, o projeto de componentes representa o software em um nível de abstração próximo do código.

Podem-se adotar três visões diferentes de projeto de componentes, dependendo da natureza do software a ser desenvolvido. A visão orientada a objetos focaliza a elaboração de classes de projeto provenientes tanto do domínio do problema como de infraestrutura. A visão tradicional refina três tipos diferentes de componentes ou módulos: módulos de controle, módulos do domínio do problema e os módulos de infraestrutura. Em ambos os casos, são aplicados conceitos e princípios básicos de projeto que levam a um software de alta qualidade. Quando considerados do ponto de vista de processos, o projeto de componentes faz uso de componentes de software reutilizáveis e padrões de projeto que são elementos fundamentais da engenharia de software baseada em componentes.

¹² As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

Uma série de importantes princípios e conceitos orienta o projetista à medida que as classes são elaboradas. Ideias englobadas pelo princípio do aberto-fechado e da inversão da dependência, além de conceitos como acoplamento e coesão, orientam o engenheiro de software na construção de componentes de software que podem ser testados, implementados e mantidos. Para conduzir projetos de componentes nesse contexto, são elaboradas classes através da especificação de detalhes de mensagens, identificação de interfaces apropriadas, elaboração de atributos e definição de estruturas de dados para implementá-las, descrevendo o fluxo de processamento em cada operação e representando o comportamento em termos de componentes ou classes. Em todos os casos, a iteração (refatoração) de projeto é uma atividade essencial.

O projeto de componentes tradicional requer a representação de estruturas de dados, interfaces e algoritmos para um módulo de programa em detalhes suficientes para orientar-nos na geração de código-fonte em uma linguagem de programação. Para tanto, o projetista usa uma série de notações de projeto que representam detalhes no nível de componentes, em formato gráfico, tabular ou com base em texto.

O projeto de componentes para WebApps considera tanto o conteúdo quanto a funcionalidade, já que é fornecido por um sistema baseado na Web. O projeto de conteúdo no nível de componentes focaliza os objetos de conteúdo e a maneira pela qual podem ser empacotados para apresentação ao usuário final de uma WebApp. O projeto funcional para WebApps concentra-se nas funções de processamento que manipulam conteúdo, efetuam cálculos, consultas e acesso a um banco de dados, bem como estabelecem interfaces com outros sistemas. Todos os princípios e diretrizes de projeto de componentes se aplicam.

Programação estruturada é uma filosofia de projeto procedural que restringe o número e o tipo de construções lógicas usadas para representar detalhes algorítmicos. O intuito da programação estruturada é auxiliar o projetista na definição de algoritmos que sejam menos complexos e, conseqüentemente, mais fáceis de ser lidos, testados e mantidos.

A engenharia de software baseada em componentes identifica, constrói, cataloga e dissemina um conjunto de componentes de software em determinado domínio de aplicação. Esses componentes são então qualificados, adaptados e integrados para uso em um novo sistema. Os componentes reutilizáveis devem ser projetados em ambiente que estabeleça estruturas de dados, protocolos de interface e arquiteturas de programa-padrão para cada domínio de aplicação.

PROBLEMAS E PONTOS A PONDERAR

10.1. O termo *componente* é, algumas vezes, difícil de definir. Forneça inicialmente uma definição genérica e, a seguir, definições mais explícitas para software tradicional e orientado a objetos. Por fim, escolha três linguagens de programação com as quais você esteja familiarizado e ilustre como cada uma define um componente.

10.2. Por que os componentes de controle são necessários em software tradicional e em geral não são em software orientado a objetos?

10.3. Descreva o OCP em suas próprias palavras. Por que é importante criar abstrações que sirvam como interface entre componentes?

10.4. Descreva o DIP em suas próprias palavras. O que poderia acontecer se um projetista criasse uma dependência muito grande nas concretizações?

10.5. Selecione três componentes que você tenha desenvolvido recentemente e avalie os tipos de coesão que cada um apresenta. Caso tivesse de definir o principal benefício da coesão elevada, qual seria?

10.6. Selecione três componentes que você tenha desenvolvido recentemente e avalie os tipos de acoplamento que cada um apresenta. Caso tivesse de definir o principal benefício de um baixo acoplamento, qual seria?

10.7. É razoável dizer que componentes de domínio do problema jamais devem apresentar acoplamento externo? Caso concorde, que tipos de componentes apresentariam acoplamento externo?

10.8. Desenvolva: (1) uma classe de projeto elaborada, (2) descrições de interface, (3) um diagrama de atividades para uma das operações contidas na classe e (4) um diagrama de estados detalhado para uma das classes do *CasaSegura* já discutidas em capítulos anteriores.

10.9. Refinamento gradual e refatoração são a mesma coisa? Em caso negativo, em que diferem?

10.10. O que é um componente de WebApp?

10.11. Selecione um pequeno trecho de um programa existente (aproximadamente 50 a 75 linhas de código-fonte). Isole as construções de programação estruturadas desenhando retângulos em torno delas no código-fonte. O trecho de programa contém construções que violam a filosofia da programação estruturada? Em caso positivo, redesenhe o código para que fique de acordo com as construções de programação estruturadas. Em caso negativo, o que você notou em relação aos retângulos que desenhou?

10.12. Todas as linguagens de programação modernas implementam as construções de programação estruturada. Forneça exemplos de três linguagens de programação.

10.13. Selecione um pequeno componente codificado e represente-o usando: (1) um diagrama de atividades, (2) um fluxograma, (3) uma tabela de decisão e (4) PDL.

10.14. Por que a ideia de “agrupamento” é importante durante o processo de revisão em projetos de componentes?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Foram publicados muitos livros sobre desenvolvimento baseado em componentes e reutilização de componentes nos últimos anos. Apperly e seus colegas (*Service- and Component-Based Development*, Addison-Wesley, 2003), Heineman e Councill (*Component Based Software Engineering*, Addison-Wesley, 2001), Brown (*Large Scale Component-Based Development*, Prentice-Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000), Herzum e Sims (*Business Component Factory*, Wiley, 1999), Allen, Frost e Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective*, Cambridge University Press, 1998) abordam todos os aspectos importantes do processo de CBSE. Cheesman e Daniels (*UML Components*, Addison-Wesley, 2000) discutem a CBSE com ênfase na UML.

Gao e seus colegas (*Testing and Quality Assurance for Component-Based Software*, Artech House, 2006) e Gross (*Component-Based Software Testing with UML*, Springer, 2005) apresentam questões relacionadas a testes e garantia da qualidade de software para sistemas baseados em componentes.

Chegaram ao mercado dezenas de livros que descrevem os padrões de mercado baseados em componentes nos últimos anos. Estes tratam dos detalhes de funcionamento dos padrões em si, mas também consideram diversos tópicos importantes da CBSE.

O trabalho de Linger, Mills e Witt (*Structured Programming — Theory and Practice*, Addison-Wesley, 1979) se mantém como um verdadeiro tratado sobre o assunto. O texto contém uma ótima PDL, bem como discussões detalhadas sobre as ramificações da programação estruturada. Outros livros que focalizam as questões procedurais para sistemas tradicionais seriam os de Robertson (*Simple Program Design*, 3. ed., Course Technology, 2000), Farrell (*A Guide to Programming Logics and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, 2. ed., Addison-Wesley, 1999) e Dahl (*Structured Programming*, Academic Press, 1997).

Relativamente poucos livros recentes têm se dedicado ao projeto de componentes. Em geral, livros de linguagens de programação tratam o projeto procedural com certo nível de detalhe, mas sempre no contexto da linguagem introduzida pelo livro. Centenas de títulos desse tipo estão disponíveis.

Uma ampla gama de fontes de informação sobre projeto de componentes se encontra disponível na Internet. Uma lista atualizada de referências na Web, relevante para o projeto de componentes, pode ser encontrada no site www.mhhe.com/engcs/comp/pressman/professional/olc/ser.htm.

PROJETO DE INTERFACES DO USUÁRIO

CONCEITOS-CHAVE

acessibilidade	305
análise de tarefas ...	295
análise de usuários ..	294
atribuição de nomes a comandos	304
avaliação de projetos	311
carga de memória ...	289
controle	288
elaboração de tarefas.	296
interface	
análise	294
consistência	290
modelos	291
projeto	300
internacionalização ..	305
princípios e diretrizes .	306
processo	292
projeto de interface para WebApp	306
recursos de ajuda ...	303
regras de ouro	288
tempo de resposta ..	302
tratamento de erros .	304
usabilidade	291

Vivemos em um mundo de produtos de alta tecnologia e praticamente todos — produtos eletrônicos de consumo, equipamentos industriais, sistemas corporativos, sistemas militares, software para PC e WebApps — requerem interação humana. Para que um produto de software seja bem-sucedido, deve apresentar boa usabilidade — medida qualitativa da facilidade e eficiência com a qual um ser humano consegue empregar as funções e os recursos oferecidos pelo produto de alta tecnologia.

Independentemente de uma interface ter sido projetada para um aparelho de música digital ou para um sistema de controle de armamento de um caça, a usabilidade importa. Se os mecanismos de interface tiverem sido bem projetados, o usuário flui suavemente através da interação usando um ritmo cadenciado que permite que o trabalho seja realizado sem grandes esforços. Entretanto, se a interface for mal concebida, o usuário se move aos trancos e barrancos, e o resultado será frustração e baixa eficiência no trabalho.

Nas primeiras três décadas da era computacional, a usabilidade não era uma preocupação dominante entre aqueles que construíam software. Em seu clássico livro sobre projeto, Donald Norman [Nor88] argumentou que já era tempo de uma mudança de atitude:

Para criar tecnologia que se adapte ao ser humano, é necessário estudá-lo. Mas hoje temos uma tendência de estudar apenas a tecnologia. Como consequência, exige-se que as pessoas se adaptem à tecnologia. É chegada a hora de inverter a tendência, a hora de fazer com que a tecnologia se adapte às pessoas.

À medida que os tecnólogos passaram a estudar a interação humana, surgiram duas questões preponderantes. Primeiro, identificou-se um conjunto de *regras de ouro* (discutidas na Seção 11.1). Estas se aplicavam a toda interação humana com produtos tecnológicos. Em segundo lugar, definiu-se um conjunto de *mecanismos de interação* para permitir aos projetistas de software construir sistemas que implementassem de forma apropriada

PANORAMA

O que é? O projeto de interfaces do usuário cria um meio de comunicação efetivo entre o ser humano e o computador. Seguindo-se um conjunto de princípios de projeto de interfaces, o projeto identifica objetos e ações de interface e então cria um layout de tela que forma a base para um protótipo de interface do usuário.

Quem realiza? Um engenheiro de software projeta a interface do usuário por meio da aplicação de um processo iterativo que faz uso de princípios de projeto predefinidos.

Por que é importante? Se um software for difícil de ser utilizado, se ele o compele a incorrer em erros ou frustra seus esforços de atingir suas metas, você não gostará dele, independentemente do poder computacional apresentado, do conteúdo fornecido ou da funcionalidade oferecida. A interface deve ser correta, pois molda a percepção do software pelo usuário.

Quais são as etapas envolvidas? O projeto de interfaces do usuário se inicia pela identificação do

usuário, das tarefas e dos requisitos do ambiente. Uma vez que as tarefas de usuário tenham sido identificadas, os cenários são criados e analisados para definir um conjunto de objetos e ações de interface. Os cenários formam a base para a criação do layout da tela que representa o projeto gráfico e o posicionamento de ícones, a definição de texto descritivo na tela, a especificação e os títulos de janelas, bem como a especificação de itens de menu principais e secundários. São usadas ferramentas para criar protótipos e, por fim, implementar o modelo de projeto e o resultado é avaliado em termos de qualidade.

Qual é o artefato? São criados cenários de usuário e gerados layouts de tela. É desenvolvido um protótipo de interface modificado de forma iterativa.

Como garantir que o trabalho foi realizado corretamente? Os usuários fazem um *test drive* da interface e o feedback do *test drive* é usado para a próxima modificação iterativa do protótipo.

as regras de ouro. Os mecanismos de interação, coletivamente denominado *interface gráfica do usuário* (*graphical user interface, GUI*), eliminaram parte da maioria dos atrozes problemas associados às interfaces humanas. Mas mesmo no “mundo Windows”, todos nós já encontramos interfaces do usuário que são difíceis de assimilar, difíceis de usar, confusas, contraintuitivas, implacáveis e, em muitos casos, totalmente frustrantes. Mesmo assim, alguns gastam tempo e energia construindo cada uma dessas interfaces e é pouco provável que seus construtores tenham criado os problemas propositadamente.

11.1 AS REGRAS DE OURO

Em seu livro sobre projeto de interfaces, Theo Mandel [Man97] cunha três *regras de ouro*:

1. Deixar o usuário no comando.
2. Reduzir a carga de memória do usuário.
3. Tornar a interface consistente.

Essas regras formam, na verdade, a base para um conjunto de princípios para o projeto de interfaces do usuário que orienta esse importante aspecto do projeto de software.

11.1.1 Deixar o usuário no comando

Durante uma sessão para levantamento de requisitos para um importante e novo sistema de informação, perguntou-se a um usuário-chave sobre os atributos da interface gráfica orientada a janelas.

“O que realmente gostaria”, disse o usuário solenemente, “é de um sistema que leia minha mente. Ele saberia o que eu quero fazer antes mesmo de eu ter de fazê-lo e isso me facilitaria tremendamente a vida. Isso é tudo, apenas isso.”

Minha primeira reação foi de sacudir a cabeça e sorrir, porém, refleti por um momento. Não havia absolutamente nada de errado com a solicitação do usuário. Ele queria um sistema que reagisse às suas necessidades e o ajudasse a concretizar suas tarefas. Ele queria controlar o computador, e não que o computador o controlasse.

A maioria das limitações e restrições de interface impostas por um projetista destina-se a simplificar o modo de interação. Mas para quem?

Como projetista, talvez sejamos tentados a introduzir restrições e limitações para simplificar a implementação da interface. O resultado poderia ser uma interface fácil de ser construída, mas frustrante sob o ponto de vista do usuário. Mandel [Man97] define uma série de princípios de projeto que permitem a um usuário manter o controle:

Defina modos de interação para não forçar o usuário a realizar ações desnecessárias ou indesejadas. Modo de interação é o estado atual da interface. Por exemplo, se for escolhido o comando de *correção ortográfica* no menu de um processador de texto, o software entra no modo de correção ortográfica. Não há nenhuma razão para forçar o usuário a permanecer no modo de revisão ortográfica se ele quer apenas fazer uma pequena edição de texto no meio do caminho. O usuário deve ser capaz de entrar e sair desse modo com pouco ou nenhum esforço.

Proporcione interação flexível. Pelo fato de diferentes usuários terem preferências de interação diversas, deve-se fornecer opções. Por exemplo, um software poderia permitir a um usuário interagir por meio de comandos de teclado, movimentação do mouse, caneta digitalizadora, tela multitoque ou por comandos de reconhecimento de voz. Mas nem toda ação é suscetível a todo mecanismo de interação. Considere, por exemplo, a dificuldade de usar comandos via teclado (ou entrada de voz) para desenhar uma forma complexa.

Possibilite que a interação de usuário possa ser interrompida e desfeita. Mesmo quando envolvido em uma sequência de ações, o usuário deve ser capaz de interromper a sequência para fazer alguma outra coisa (sem perder o trabalho que já havia feito). O usuário também deve ser capaz de “desfazer” qualquer ação.

“É melhor projetar a experiência do usuário do que retificá-la.”

Jon Meads

“Sempre desejei que meu computador fosse tão fácil de ser usado quanto meu telefone. Meu desejo tornou-se realidade. Já não sei mais como usar meu telefone.”

Bjarne Stroustrup
(criador do C++)

Simplifique a interação à medida que os níveis de competência avançam e permita que a interação possa ser personalizada. Geralmente os usuários constatarem que realizam a mesma sequência de interações repetidamente. Vale a pena criar um mecanismo de “macros” que permita a um usuário avançado personalizar a interface para facilitar sua interação.

Oculte os detalhes técnicos de funcionamento interno do usuário casual. Uma interface com o usuário deve levá-lo ao mundo virtual da aplicação. O usuário não deve se preocupar com o sistema operacional, as funções de arquivos ou alguma outra tecnologia computacional enigmática. Em essência, a interface jamais deve exigir que o usuário interaja em um nível “interno” à máquina (por exemplo, jamais se deve exigir que um usuário tenha de digitar comandos do sistema operacional a partir do ambiente da aplicação).

Projete para interação direta com objetos que aparecem na tela. O usuário tem uma sensação de controle quando lhe é permitido manipular os objetos necessários para realizar uma tarefa de maneira similar ao que ocorreria caso o objeto fosse algo físico. Por exemplo, uma interface de aplicação que permita a um usuário “esticar” um objeto (aumentar ou diminuir em escala o seu tamanho) é uma implementação de manipulação direta.

11.1.2 Reduzir a carga de memória do usuário

Quanto mais um usuário tiver de se lembrar, mais sujeita a erros será a interação com o sistema. É por essa razão que uma interface do usuário bem desenhada não exaure a memória do usuário. Sempre que possível, o sistema deve “se lembrar” de informações pertinentes e auxiliar o usuário em um cenário de interação que o ajude a recordar-se. Mandel [Man97] define princípios de projeto que possibilitam a uma interface reduzir a carga de memória do usuário:

Reduza a demanda de memória recente. Quando os usuários estão envolvidos em tarefas complexas, a demanda de memória recente pode ser significativa. A interface deve ser projetada para reduzir a exigência de recordar ações, entradas e resultados passados. Isso pode ser obtido pelo fornecimento de pistas visuais que permitam a um usuário reconhecer ações passadas, em vez de ter de se recordar delas.

Estabeleça defaults significativos. O conjunto de parâmetros iniciais (defaults) deve fazer sentido para o usuário comum, porém um usuário também deve ser capaz de especificar suas preferências individuais. Entretanto, deve-se fornecer uma opção “reset”, que permita o restabelecimento dos valores-padrão originais.

Defina atalhos intuitivos. Quando forem usados mnemônicos para realizar alguma função de sistema (por exemplo, alt-P para chamar a função de impressão), esse mnemônico deve estar ligado à ação de uma forma que seja fácil de ser memorizada (por exemplo, a primeira letra da tarefa a ser solicitada).

O layout visual da interface deve se basear na metáfora do mundo real. Por exemplo, um sistema de pagamento de contas deve usar uma metáfora de talão de cheques e registro de cheques para orientar o usuário pelo processo de pagamento de uma conta. Isso permite ao usuário que se apoie em indicações visuais bem compreensíveis, em vez de ter de memorizar uma sequência de interações misteriosa.

Revele as informações de maneira progressiva. A interface deve ser organizada hierarquicamente. As informações sobre uma tarefa, um objeto ou algum comportamento devem ser apresentadas inicialmente em um alto nível de abstração. Mais detalhes devem ser apresentados após o usuário demonstrar interesse por meio de uma seleção com o mouse. Um exemplo, comum a muitas aplicações de processamento de texto, é a função de sublinhado. A função em si é uma de uma série de funções agrupadas em um menu de *estilo de texto*. Entretanto, todos os recursos de sublinhado não são apresentados logo de cara. O usuário tem de selecionar a função de sublinhado; em seguida, todas as opções de sublinhado (por exemplo, sublinhado simples, duplo, tracejado) são apresentadas.

CASA SEGURA



Violação de uma regra de ouro de uma interface do usuário

Cena: Sala do Vinod, quando é iniciado o projeto da interface do usuário.

Atores: Vinod e Jamie, membros da equipe de engenharia de software do CasaSegura.

Conversa:

Jamie: Estive pensando sobre a interface da função de vigilância.

Vinod (sorrindo): Pensar faz bem.

Jamie: Creio que talvez possamos simplificar um pouco as coisas.

Vinod: O que você quer dizer?

Jamie: Bem, que tal se eliminássemos totalmente a planta. Ela é “atraente”, porém irá nos dar muito trabalho de projeto. Em vez disso, poderíamos apenas solicitar ao usuário especificar a câmera que deseja ver e então exibir o vídeo em uma janela de vídeo.

Vinod: Como o proprietário irá se lembrar de quantas câmeras estão configuradas e onde se encontram?

Jamie (levemente irritado): Ele é o proprietário do imóvel; ele deve saber essas coisas.

Vinod: Mas e se não souber?

Jamie: Ele deve.

Vinod: Essa não é a questão... E se ele se esquecer?

Jamie: Poderíamos fornecer-lhe uma lista de câmeras em operação e suas posições.

Vinod: Isso é possível, mas por que ele deveria ter de solicitar uma lista?

Jamie: Certo, nós fornecemos a lista se ele pedir ou não.

Vinod: Melhor. Pelo menos não terá de se lembrar de coisas que podemos dar a ele.

Jamie (pensando por um instante): Mas você gosta da planta, não é mesmo?

Vinod: Sim.

Jamie: Qual delas você acha que o pessoal de marketing irá gostar?

Vinod: Tá brincando, não é mesmo?

Jamie: Não.

Vinod: Sei lá... Aquela com efeito piscante... Eles adoram características chamativas para o produto... Eles não estão interessados em qual é a mais fácil de ser construída.

Jamie (suspirando): Certo, talvez possamos fazer um protótipo para ambas.

Vinod: Boa ideia... Depois deixamos o cliente decidir.

11.1.3 Tornar a interface consistente

“Coisas que parecem diferentes deveriam agir distintamente. Coisas que parecem iguais deveriam agir da mesma forma.”

Larry Marine

A interface deve apresentar e obter informações de forma consistente. Isso implica: (1) todas as informações visuais são organizadas de acordo com regras de projeto mantidas ao longo de todas as exibições de telas, (2) mecanismos de entrada são restritos a um conjunto limitado que é usado de forma consistente por toda a aplicação e (3) mecanismos de navegação para passar de uma tarefa a outra são definidos e implementados de maneira consistente. Mandel [Man97] define um conjunto de princípios de projeto que ajudam a tornar a interface consistente:

Permita ao usuário inserir a tarefa atual em um contexto significativo. Muitas interfaces implementam camadas de interações complexas com dezenas de imagens de tela. É importante fornecer indicadores (por exemplo, títulos para as janelas, ícones gráficos, sistema de cores consistente) que possibilitem ao usuário saber o contexto do trabalho em mãos. Além disso, o usuário deve ser capaz de determinar de onde ele veio e quais alternativas existem para transição para uma nova tarefa.

Mantenha a consistência ao longo de uma família de aplicações. Um conjunto de aplicações (ou produtos) deve implementar as mesmas regras de projeto de modo que a consistência seja mantida para toda a interação.

Se modelos interativos anteriores tiverem criado expectativa nos usuários, não faça alterações a menos que haja uma forte razão para isso. Uma vez que determinada sequência interativa tenha se tornado um padrão de fato (por exemplo, o uso de alt-S para salvar um arquivo), o usuário pressupõe que isso vá ocorrer em qualquer aplicação que vá usar. Uma mudança (por exemplo, usar alt-S para chamar uma função de escala) irá causar confusão.

Os princípios de projeto para interfaces discutidos nesta e em seções anteriores dão uma orientação básica. Nas seções seguintes, veremos o processo de projeto de interfaces em si.

INFORMAÇÕES

**Usabilidade**

Em um artigo esclarecedor sobre usabilidade, Larry Constantine [Con95] faz uma pergunta que tem uma relevância significativa sobre o tema: “O que os usuários querem, afinal de contas?”. Ele responde da seguinte maneira:

O que os usuários realmente querem são boas ferramentas. Todos os sistemas de software, de sistemas operacionais e linguagens a aplicações de entrada de dados e de apoio à decisão, são apenas ferramentas. Os usuários finais querem das ferramentas que criamos para eles praticamente o mesmo que esperamos das ferramentas que utilizamos. Eles querem sistemas fáceis de aprender e que os ajude a realizar seu trabalho. Querem software que não os retarde, não os engane ou confunda, que não facilite a prática de erros ou dificulte a finalização de seus trabalhos.

Constantine argumenta que a usabilidade não é derivada da estética, mecanismos de interação de última geração ou de inteligência incorporada às interfaces. Ao contrário, ocorre quando a arquitetura da interface atende às necessidades das pessoas que a usarão.

Uma definição formal de usabilidade é um tanto ilusória. Donahue e seus colegas [Don99] a definem da seguinte maneira: “Usabilidade é uma medida do quanto um sistema computacional... Facilita o aprendizado; ajuda os aprendizes a se lembrarem daquilo que aprenderam; reduz a probabilidade de erros; permite que se tornem eficientes e os deixa satisfeitos com o sistema”.

A única maneira de determinar se existe ou não “usabilidade” em um sistema que estamos construindo é realizar a ava-

liação ou teste de usabilidade. Observe os usuários interagirem com o sistema e faça-lhes as seguintes perguntas [Con95]:

- O sistema é utilizável sem a necessidade de ajuda ou aprendizado contínuo?
- As regras de interação ajudam um usuário experiente a trabalhar eficientemente?
- Os mecanismos de interação se tornam mais flexíveis à medida que os usuários se tornam mais capacitados?
- O sistema foi ajustado para o ambiente físico e social em que será usado?
- O usuário está ciente do estado do sistema? O usuário sempre sabe onde se encontra?
- A interface é estruturada de maneira lógica e consistente?
- Os mecanismos de interação, ícones e procedimentos são consistentes por toda a interface?
- A interação antecipa erros e ajuda o usuário a corrigi-los?
- A interface é tolerante com erros cometidos?
- A interação é simples?

Se cada uma dessas questões for respondida “positivamente”, é provável que a usabilidade tenha sido atingida.

Entre os muitos benefícios mensuráveis obtidos de um sistema utilizável, temos [Don99]: aumento nas vendas e na satisfação do cliente, vantagem competitiva, melhores avaliações por parte da mídia, melhor recomendação boca a boca, menores custos de suporte, aumento da produtividade do usuário final, redução nos custos de treinamento e de documentação, menor probabilidade de litígio com clientes descontentes.

11.2 ANÁLISE E PROJETO DE INTERFACES

WebRef

Uma excelente fonte de informações sobre projeto de interfaces do usuário pode ser encontrada em www.useit.com.

O processo geral para análise e projeto de uma interface do usuário se inicia com a criação de diferentes modelos de funções do sistema (segundo uma percepção do mundo externo). Começamos pelo delineamento das tarefas orientadas à interação homem-máquina necessárias para alcançar a função do sistema e, em seguida, consideramos as questões de projeto que se aplicam a todos os projetos de interface. São usadas ferramentas para prototipagem e, por fim, a implementação do modelo de projeto para o resultado ser avaliado pelos usuários finais em termos de qualidade.

11.2.1 Modelos de análise e projeto de interfaces

Quatro modelos distintos entram em cena ao se analisar e projetar uma interface do usuário. Um engenheiro humano (ou o engenheiro de software) estabelece um *modelo de usuário*, o engenheiro de software cria um *modelo de projeto*, o usuário final desenvolve uma imagem mental em geral chamada *modelo mental* do usuário ou *percepção do sistema* e os implementadores do sistema criam um *modelo de implementação*. Infelizmente, cada um dos modelos pode diferir significativamente. O papel de um projetista de interfaces é reconciliar as diferenças e obter uma representação consistente da interface.

O modelo de usuário estabelece o perfil dos usuários finais do sistema. Em sua coluna introdutória sobre “projeto centralizado no usuário”, Jeff Patton [Pat07] observa o seguinte:

“Se há um ‘truque’ para isso, a interface do usuário está quebrada.”

Douglas
Anderson



Até mesmo um usuário novato quer atalhos; mesmo usuários frequentes e com conhecimentos algumas vezes precisam de orientação. Dê a eles aquilo de que precisam.

PONTO-CHAVE

O modelo mental do usuário molda como o usuário percebe a interface e se a interface do usuário atende às suas necessidades.

“Preste atenção naquilo que os usuários fazem e não no que dizem.”

Jakob Nielsen

A verdade é que, projetistas e desenvolvedores — até eu mesmo — em geral pensam nos usuários. Entretanto, na ausência de um consistente modelo mental de usuário específico, nos colocamos no lugar desses usuários. A autossustituição não é centralizada no usuário — é egocêntrica.

Para construir uma interface do usuário efetiva, “todo projeto deve começar com um entendimento dos usuários pretendidos, incluindo seus perfis de idade, gênero, habilidades físicas, educação, formação cultural ou origem étnica, motivação, metas e personalidade” [Shn04]. Além disso, os usuários podem ser classificados como:

Novatos. Nenhum conhecimento sintático¹ do sistema e pouco conhecimento semântico² da aplicação ou uso do computador em geral.

Usuários intermitentes e com conhecimento. Razoável conhecimento semântico da aplicação, mas lembrança relativamente baixa das informações sintáticas necessárias para usar a interface.

Usuários frequentes e com conhecimento. Bom conhecimento semântico e sintático que muitas vezes levam à “síndrome do usuário poderoso”; indivíduos que procuram atalhos e modos de interação abreviados.

O *modelo mental* de usuário (percepção do sistema) é a imagem do sistema que os usuários finais trazem em suas mentes. Por exemplo, se fosse perguntado ao usuário de determinado processador de texto para descrever sua operação, a percepção do sistema orientaria a resposta. A precisão da descrição irá depender do perfil do usuário (por exemplo, novatos dariam no máximo uma resposta muito superficial) e da familiaridade geral com o software no domínio de aplicação. Um usuário que entenda completamente de processadores de texto, mas que trabalhou com um processador de texto específico apenas uma vez, talvez, na verdade, fosse capaz de dar uma descrição mais completa de sua função do que o novato que investiu semanas tentando aprender o sistema.

O *modelo de implementação* combina a manifestação externa do sistema computacional (a aparência e a percepção da interface), acoplada a todas as informações de apoio (livros, manuais, fitas de vídeo, arquivos de ajuda) que descrevem a sintaxe e a semântica da interface. Quando o modelo de implementação e o modelo mental de usuário são coincidentes, em geral os usuários sentem-se à vontade com o software e o utilizam de maneira eficaz. Para conseguir tal “amalgama” dos modelos, o modelo de projeto deve ter sido desenvolvido para levar em conta as informações contidas no modelo de usuário, e o modelo de implementação deve refletir precisamente as informações sintáticas e semânticas sobre a interface.

Os modelos descritos nesta seção são “abstrações daquilo que o usuário está fazendo ou pensa que está fazendo ou aquilo que alguém pensa que deveria estar fazendo quando usa um sistema interativo” [Mon84]. Em essência, esses modelos possibilitam que o projetista de interfaces satisfaça um elemento-chave do mais importante princípio do projeto de interfaces do usuário: “Conhecendo o usuário, você conhecerá as tarefas”.

11.2.2 O processo

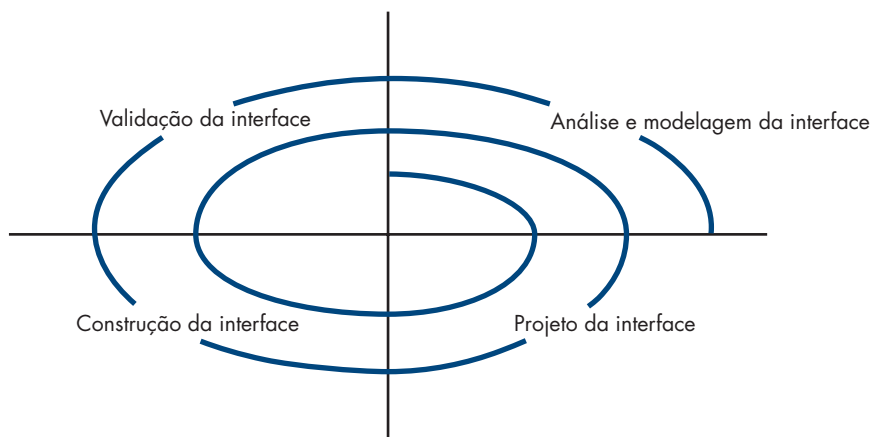
O processo de análise e projeto para interfaces do usuário é iterativo e pode ser representado usando-se um modelo espiral similar ao discutido no Capítulo 2. De acordo com a Figura 11.1, o processo de análise e projeto de interfaces do usuário começa no interior da espiral e engloba quatro atividades estruturais distintas [Man97]: (1) análise e modelagem de interfaces, (2) projeto de interfaces, (3) construção de interfaces e (4) validação de interfaces. A espiral da Figura 11.1 implica que cada uma dessas tarefas ocorrerá mais de uma vez; cada volta em torno da espiral representa a elaboração adicional dos requisitos e do projeto resultante. Na maioria

¹ Nesse contexto, *conhecimento sintático* refere-se à mecânica de interação necessária para usar a interface de forma efetiva.

² *Conhecimento semântico* refere-se ao sentido subjacente da aplicação — um entendimento das funções realizadas, a medida de entrada e saída e as metas e objetivos do sistema.

FIGURA 11.1

O processo de projeto de interface do usuário



dos casos, a atividade de construção envolve prototipagem — a única maneira prática de validar o que foi projetado.

A *análise de interfaces* focaliza o perfil dos usuários que irão interagir com o sistema. O nível de habilidades, o conhecimento da área e a receptividade geral em relação ao novo sistema são registrados e categorias de usuários são definidas. Para cada categoria de usuário, é feito o levantamento de requisitos. Em essência, trabalhamos para compreender a percepção do sistema (Seção 11.2.1) para cada classe de usuário.

Uma vez definidos os requisitos gerais, é realizada uma *análise de tarefas* mais detalhada. Aquelas tarefas que o usuário realiza para alcançar os objetivos do sistema são identificadas, descritas e elaboradas (ao longo de uma série de passagens iterativas pela espiral). A análise de tarefas é discutida de forma mais detalhada na Seção 11.3. Por fim, a análise do ambiente do usuário concentra-se no ambiente de trabalho físico. Entre as perguntas a ser feitas, temos:

- Onde estará fisicamente localizada a interface?
- O usuário estará sentado, de pé ou realizando outras tarefas não relacionadas com a interface?
- O hardware de interface leva em conta restrições de espaço, luz ou ruído?
- Existem considerações de fatores humanos especiais determinados por fatores ambientais?

As informações coletadas como parte da ação de análise são usadas para criar um modelo de análise para a interface. Usando esse modelo como base, a atividade de projeto se inicia. A meta do *projeto da interface* é definir um conjunto de objetos e ações de interface (e suas representações na tela) que permitam a um usuário realizar todas as tarefas definidas para atender todas as metas de usabilidade definidas para o sistema. O projeto de interfaces é discutido de forma mais detalhada na Seção 11.4.

A *construção da interface* em geral se inicia com a criação de um protótipo que permite que cenários de uso possam ser avaliados. À medida que o processo de projeto iterativo prossegue, um kit de ferramentas de interfaces do usuário (Seção 11.5) poderia ser usado para completar a construção da interface.

A *validação da interface* se concentra: (1) na capacidade de a interface implementar corretamente todas as tarefas de usuário, levar em conta todas as variações de tarefas, bem como atender todas os requisitos gerais dos usuários; (2) no grau de facilidade de uso e aprendizado da interface; e (3) na aceitação dos usuários da interface como uma útil ferramenta no seu trabalho.

“É melhor projetar a experiência do usuário do que retificá-la.”

Jon Meads

? O que precisamos saber sobre o ambiente ao iniciarmos o projeto de interfaces do usuário?

Conforme já citado, as atividades descritas nesta seção ocorrem iterativamente. Consequentemente, não há nenhuma necessidade de tentar especificar todos os detalhes (para modelo de análise ou de projeto) na primeira passagem. Passagens subsequentes ao longo do processo elaboram detalhes de tarefas, informações de projeto e características operacionais da interface.

11.3 ANÁLISE DE INTERFACES³

Um princípio fundamental de todos os modelos de processos de engenharia de software é o seguinte: *entender o problema antes de tentar desenvolver uma solução*. No caso do projeto de interfaces do usuário, entender o problema significa entender: (1) as pessoas (usuários finais) que irão interagir com o sistema por meio da interface, (2) as tarefas que os usuários finais devem realizar para completar seus trabalhos, (3) o conteúdo que é apresentado como parte da interface e (4) o ambiente onde essas tarefas serão conduzidas. Nas seções seguintes, examinaremos cada um dos elementos da análise de interfaces com o objetivo de estabelecer uma sólida base para as tarefas de projeto que se seguem.

11.3.1 Análise de usuários

A frase “interface do usuário” provavelmente seja a única justificativa necessária para desperdirmos algum tempo para entender o usuário antes de nos preocuparmos com as questões técnicas. Citei anteriormente que cada usuário tem uma imagem mental do software que pode ser diversa da imagem mental desenvolvida por outros usuários. Além do mais, a imagem mental do usuário pode ser muito diferente do modelo de projeto do engenheiro de software. A única maneira para se fazer com que a imagem mental e o modelo de projeto convirjam é tentar entender os próprios usuários, bem como as pessoas que usam o sistema. Informações de uma ampla gama de fontes podem ser usadas para concretizar isso:

 **Como saber o que o usuário quer da interface do usuário?**

Entrevistas com os usuários. Na abordagem mais direta, membros da equipe de software se encontram com os usuários finais para entender melhor suas necessidades, motivações, cultura de trabalho e uma miríade de outras questões. Isso pode ser conseguido em reuniões um-a-um ou por grupos de sondagem.

Informações do pessoal de vendas. O pessoal de vendas se encontra regularmente com usuários e pode reunir informações que ajudarão a equipe de software a classificar os usuários e compreender melhor seus requisitos.

Informações do pessoal de marketing. A análise de mercado pode ser inestimável na definição de segmentos de mercado e no entendimento de como cada segmento poderia usar o software de modos sutilmente diferentes.

Informações do pessoal de suporte. A equipe de suporte conversa diariamente com os usuários. Eles são, provavelmente, a fonte mais provável de informações sobre o que funciona e o que não funciona, o que os usuários gostam e o que não gostam, quais recursos geram questionamentos e quais são fáceis de ser usados.

O conjunto de perguntas a seguir (adaptado de [Hac98]) irá ajudá-lo a entender melhor os usuários de um sistema:

- Os usuários são profissionais treinados, técnicos, do setor administrativo ou pessoal de fábrica?
- Que nível de educação formal o usuário médio possui?
- Os usuários são capazes de aprender por meio de material escrito ou expressaram seu desejo por um treinamento em sala de aula?



Acima de tudo, invista tempo conversando com os usuários de fato, mas seja cauteloso. Uma opinião firme não significa necessariamente que a maioria dos usuários irá concordar.

PONTO-CHAVE

Como colocar-se a par da demografia e características dos usuários finais?

³ É razoável argumentar que essa seção deveria ser colocada no Capítulo 5, 6 ou 7, já que questões de análise de requisitos são discutidas lá. Ela foi inserida aqui porque a análise e o projeto de interfaces estão intimamente ligados entre si e a fronteira entre os dois em geral é confusa.

- Os usuários são digitadores experientes ou têm fobia a teclados?
- Qual a faixa etária da comunidade de usuários?
- Os usuários serão representados predominantemente por um gênero?
- Como os usuários são recompensados pelo trabalho realizado?
- Os usuários trabalham em expediente normal ou ficam até que o trabalho seja completado?
- O software deverá ser parte integrante dos usuários de trabalho ou será usado apenas esporadicamente?
- Qual o principal idioma falado pelos usuários?
- Quais as consequências se um usuário cometer um erro ao usar o sistema?
- Os usuários são especialistas no assunto tratado pelo sistema?
- Os usuários querem saber sobre a tecnologia que se encontra por trás da interface?

Assim que essas perguntas forem respondidas, saberemos quem são os usuários finais, o que provavelmente irá motivá-los, como poderão ser agrupados em diferentes classes ou perfis, quais são seus modelos mentais do sistema e como uma interface deve ser caracterizada para atender suas necessidades.

11.3.2 Análise e modelagem de tarefas

O objetivo da análise de tarefas é responder às seguintes questões:

- Que trabalho o usuário irá realizar em circunstâncias específicas?
- Quais tarefas e subtarefas serão realizadas à medida que o usuário desenvolve seu trabalho?
- Quais os objetos de domínio de problema específicos que o usuário irá manipular à medida que o trabalho é desenvolvido?
- Qual a sequência de tarefas — o fluxo de trabalho?
- Qual a hierarquia das tarefas?

Para responder a essas questões, deve-se fazer uso das técnicas discutidas anteriormente neste livro, mas neste caso, elas são aplicadas à interface do usuário.

Casos de uso. Em capítulos anteriores vimos que um caso de uso descreve a maneira pela qual um ator (no contexto do projeto de interfaces com o usuário, um ator é sempre uma pessoa) interage com um sistema. Quando usado como parte da análise de tarefas, o caso de uso é desenvolvido para mostrar como um usuário final realiza alguma tarefa relacionada com algum trabalho específico. Na maioria das oportunidades, o caso de uso é redigido em estilo informal (um parágrafo simples) na primeira pessoa. Suponhamos, por exemplo, que uma pequena empresa de software queira construir um sistema de projeto apoiado por computador explicitamente para arquitetos de interiores. Para se ter uma ideia de como eles realizam seus trabalhos, solicita-se a arquitetos de interiores que descrevam uma função de projeto específica. Ao ser indagado “Como você decide onde colocar o mobiliário em uma sala?”, um arquiteto de interiores escreveu o seguinte caso de uso informal:

Começo esboçando a planta baixa da sala, as dimensões e a localização das janelas e portas. Preocupo-me muito com a iluminação do ambiente, com a vista das janelas (caso seja bonita, quero que sobressaia), com o comprimento total livre de uma parede, com o fluxo de movimento pelo ambiente. Em seguida, pesquiso a lista de mobiliário que eu e meu cliente escolhemos — mesas, cadeiras, sofás, armários, a lista de acessórios — lâmpadas, tapetes, pinturas, escultura, plantas, peças menores e minhas anotações sobre quaisquer desejos que meu cliente tenha a colocar. Em seguida, desenho cada item das minhas listas usando um gabarito colocado em escala na planta. Nomeio cada item que desenho e uso um lápis, pois sempre acabo mudando as coisas. Considero uma série de posições alternativas e opto por aquela que mais me agrada. Em seguida, crio um *rendering* (uma figura 3-D) do ambiente, para que meu cliente tenha uma sensação de como ele ficará.

PONTO-CHAVE

A meta do usuário é realizar uma ou mais tarefas via interface do usuário. Para tanto, a interface deve fornecer mecanismos que permitam ao usuário atingir sua meta.

Esse caso de uso dá uma descrição básica de uma importante tarefa para o sistema de projeto com auxílio de computador. Por meio dele, podemos extrair tarefas, objetos e o fluxo geral da interação. Além disso, outras características do sistema que poderiam agradar o arquiteto de interiores também poderiam ser concebidas. Por exemplo, poderíamos tirar uma foto digital da vista de cada janela do ambiente. Quando o ambiente passar pelo processo de *rendering*, a vista externa real poderia ser representada através de cada janela.



A elaboração de tarefas é bem útil, porém, também pode ser perigosa. Não pressuponha, apenas por ter elaborado uma tarefa, que não exista uma outra maneira de realizá-la e que essa outra maneira será tentada quando a interface do usuário for implementada.

Elaboração de tarefas. No Capítulo 8, discutimos a elaboração gradual (também denominada decomposição funcional ou refinamento gradual) como um mecanismo para refinar as tarefas de processamento necessárias para o software realizar alguma função desejada.

A análise de tarefas para projeto de interfaces usa uma abordagem de refinamento para auxiliar a entender as atividades humanas que uma interface do usuário deve atender. A análise de tarefas pode ser aplicada de duas formas. Conforme já citado, em geral, é usado um sistema computacional interativo para substituir uma atividade manual ou semimanual. Para compreendermos as tarefas que devem ser realizadas para atingir a meta da atividade, temos de entender as tarefas que as pessoas realizam atualmente (ao usar um método manual) e, em seguida, mapeá-las em um conjunto de tarefas similar (mas não necessariamente idêntico) implementadas no contexto de uma interface do usuário. Como alternativa, podemos estudar uma especificação existente para uma solução baseada em computador e obter um conjunto de tarefas de usuário que irá atender o modelo de usuário, o modelo de projeto e a percepção do sistema.

CASA SEGURA



Casos de uso para o projeto de interfaces do usuário

Cena: Sala do Vinod, enquanto prossegue o projeto de interface do usuário.

Atores: Vinod e Jamie, membros da equipe de engenharia de software do CasaSegura.

Conversa:

Jamie: Fiz com que nosso contato de marketing redigisse um caso de uso para a interface de vigilância.

Vinod: Do ponto de vista de quem?

Jamie: Do proprietário do imóvel, quem mais poderia ser?

Vinod: Há também o papel do administrador do sistema, mesmo que o próprio proprietário esteja desempenhando a função, é um ponto de vista diferente. O “administrador” ativa o sistema, configura as coisas, faz o layout da planta, posiciona as câmeras...

Jamie: Em suma, desempenha o papel do proprietário quando ele quiser assistir ao vídeo.

Vinod: Está bem. Esse é um dos principais comportamentos da interface da função de vigilância. Mas também teremos de examinar o comportamento da administração do sistema.

Jamie (irritado): Você está certo.

[Jamie sai em busca da pessoa de marketing. Ele retorna algumas horas mais tarde.]

Jamie: Tive sorte, encontrei-a e trabalhamos juntos no caso de uso do administrador. Basicamente, iremos definir “administra-

ção” como uma função que se aplica a todas as demais funções do CasaSegura. Eis a que conclusão chegamos.

[Jamie mostra o caso de uso informal a Vinod.]

Caso de uso informal: Quero ser capaz de estabelecer ou editar o layout do sistema a qualquer momento. Ao configurar o sistema, seleciono uma função administrativa. Ela me pergunta se quero fazer uma nova configuração ou editar uma já existente. Caso opte por uma nova configuração, o sistema exibe uma tela de desenho que me permitirá desenhar a planta em uma grade de pontos. Existirão ícones para as paredes, janelas e portas para facilitar o desenho. Tenho simplesmente que “esticar” os ícones até atingirem os comprimentos apropriados. O sistema irá mostrar os comprimentos em pés ou metros (posso escolher o sistema de medidas). Posso escolher de uma biblioteca de sensores e câmeras e colocá-los na planta. Tenho que dar nome a cada um deles ou deixar que o sistema faça automaticamente. Posso estabelecer ajustes para os sensores e câmeras por meio de menus apropriados. Caso opte por editar, poderei movimentar os sensores ou as câmeras, acrescentar novas(os) ou eliminar algum(a) existente, editar a planta, bem como os ajustes de configuração para as câmeras e sensores. Em cada um dos casos, espero que o sistema realize testes de consistência e me ajude a não cometer erros.

Vinod (após ler o cenário): Certo, provavelmente existem alguns úteis padrões de projeto [Capítulo 12] ou componentes reutilizáveis para GUIs para programas de desenho. Posso apostar 50 paus que conseguimos implementar parte ou a maior parte da interface de administrador usando-os.

Jamie: De acordo. Verificarei isso.

Independentemente da abordagem geral para a análise de tarefas, temos de, primeiramente, definir e classificar as tarefas. Já citei que uma das abordagens é a elaboração gradual. Reconsideremos, por exemplo, o sistema de projeto auxiliado por computador para arquitetos de interiores, discutido anteriormente. Observando um arquiteto de interiores em trabalho, percebemos que o projeto do interior compreende uma série de atividades principais: layout do mobiliário (note o caso de uso discutido anteriormente), escolha de tecidos e materiais, escolha de papéis de parede e acabamentos para janelas, apresentação (para o cliente), custo e compras. Cada uma das tarefas principais pode ser elaborada em subtarefas. Por exemplo, usando informações contidas no caso de uso, o layout de mobiliário pode ser refinado nas seguintes tarefas: (1) desenhar uma planta nas dimensões do ambiente, (2) colocar as janelas e as portas nos locais apropriados, (3a) usar gabaritos de mobiliário para desenhar contornos de mobiliário em escala na planta, (3b) usar gabaritos de acessórios para desenhar acessórios em escala na planta, (4) movimentar contornos de mobiliário e de acessórios para obter o melhor posicionamento, (5) identificar todos os contornos de mobiliário e acessórios, (6) indicar as dimensões para mostrar as posições e (7) desenhar uma vista em perspectiva para o cliente. Poderia ser usada uma abordagem similar para cada uma das principais tarefas.

As subtarefas 1 a 7 podem ser refinadas ainda mais. As subtarefas 1 a 6 serão realizadas por meio da manipulação das informações e realização de ações em uma interface do usuário. Por outro lado, a subtarefa 7 pode ser realizada automaticamente no software e resultará pouca interação⁴ direta com o usuário. O modelo de projeto da interface deve considerar cada uma das tarefas de maneira consistente com o modelo de usuário (o perfil de um “típico” arquiteto de interiores) e a percepção do sistema (o que o arquiteto de interiores espera de um sistema automatizado).

Elaboração de objetos. Em vez de nos concentrarmos nas tarefas que um usuário tem de realizar, podemos examinar o caso de uso e outras informações obtidas do usuário e extrair os objetos físicos utilizados pelo arquiteto de interiores. Tais objetos podem ser classificados em classes. São definidos os atributos de cada classe e uma avaliação das ações aplicadas a cada objeto fornece uma lista de operações. Por exemplo, o gabarito de mobiliário talvez pudesse ser traduzido em uma classe chamada **Mobiliário** com atributos que poderiam incluir **tamanho, forma, posição** e outros. O arquiteto de interiores *selecionaria* o objeto da classe **Mobiliário**, o *moveria* para uma posição na planta (um outro objeto nesse contexto), *desenharia* o contorno da móvel e assim por diante. As tarefas *selecionar*, *mover* e *desenhar* são operações. O modelo de análise da interface do usuário não forneceria uma implementação literal para cada uma dessas operações. Entretanto, à medida que o projeto é elaborado, os detalhes das operações são definidos.

Análise do fluxo de trabalho. Quando uma série de usuários, desempenhando diferentes papéis, faz uso de uma interface do usuário, algumas vezes se faz necessário ir além da análise de tarefas e da elaboração dos objetos e aplicar a *análise do fluxo de trabalho*. Essa técnica permite que entendamos como um processo de trabalho é completado quando várias pessoas (e papéis) estão envolvidas. Consideremos uma empresa que pretenda automatizar completamente o processo de prescrição e entrega de medicamentos. Todo o processo⁵ irá girar em torno de uma aplicação baseada na Web que pode ser acessada por médicos (ou seus assistentes), farmacêuticos e pacientes. O fluxo de trabalho pode ser representado efetivamente por um diagrama de raias UML (uma variação do diagrama de atividades).

Consideramos apenas uma pequena parte do processo de trabalho: a situação que ocorre quando um paciente solicita uma revalidação da receita. A Figura 11.2 apresenta um diagrama de raias que indica as tarefas e decisões para cada um dos três papéis citados anteriormente.



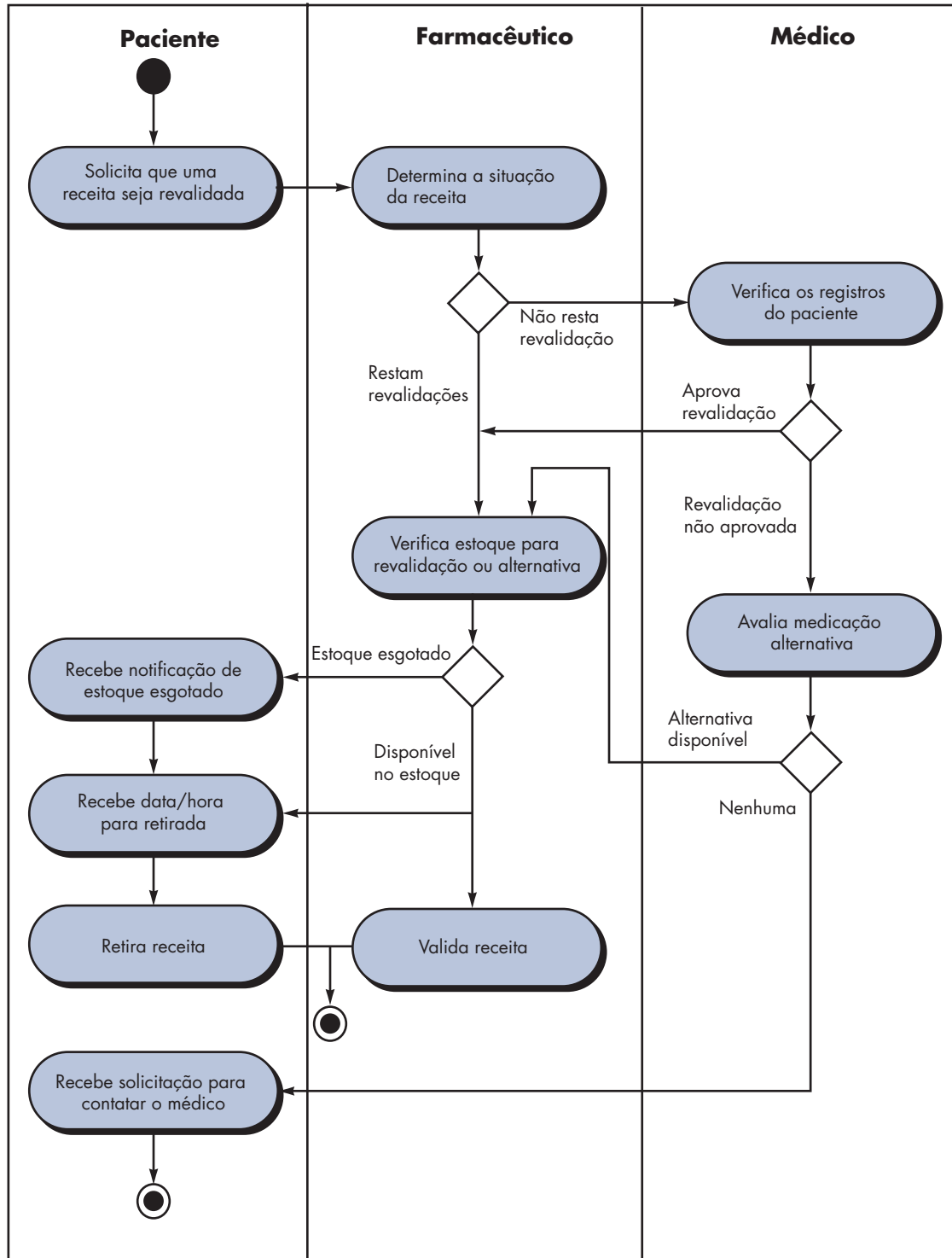
Embora a elaboração de objetos seja útil, não deve ser usada como um método isolado. A opinião do usuário tem de ser considerada durante a análise de tarefas.

⁴ Entretanto, talvez esse não seja o caso. O arquiteto de interiores talvez queira especificar a perspectiva a ser desenhada, a escala, o uso de cores e outras informações. O caso de uso relacionado ao desenho de vistas em perspectiva com efeito de rendering forneceria as informações necessárias para lidar com essa tarefa.

⁵ Esse exemplo foi adaptado de [Hac98].

FIGURA 11.2

Diagrama de raias para a função de revalidação de receita



As informações talvez sejam levantadas via entrevista ou por meio de casos de uso escritos por cada ator. Independentemente disso, o fluxo de eventos (mostrado na figura) nos permite reconhecer uma série de características-chave da interface:

“É muito melhor adaptar a tecnologia ao usuário do que forçá-lo a se adaptar à tecnologia.”

Larry Marine

1. Cada usuário implementa tarefas distintas via interface; consequentemente, o aspecto da interface projetada para o paciente será diferente do definido para os farmacêuticos ou médicos.
2. O projeto da interface para os farmacêuticos e médicos deve contemplar acesso e exibição de informações de fontes de informação secundárias (por exemplo, acesso ao estoque para o farmacêutico e acesso a informações sobre medicamentos alternativos para o médico).
3. Muitas das atividades citadas no diagrama de raiais podem ser elaboradas ainda mais com o uso de análise de tarefas e/ou elaboração de objetos (por exemplo, *Preenche receita* poderia implicar uma entrega via correio, uma ida à farmácia ou a um centro de distribuição de medicamentos especial).

Representação hierárquica. Um processo de elaboração ocorre quando se começa a analisar a interface. Uma vez estabelecido o fluxo de trabalho, pode ser definida uma hierarquia de tarefas para cada tipo de usuário. A hierarquia é obtida por meio de uma elaboração gradual de cada tarefa identificada para o usuário. Consideremos, por exemplo, a seguinte tarefa de usuário e sub-hierarquia de tarefas.

Tarefa de usuário: *Solicita que uma receita seja revalidada*

- *Fornecer informações de identificação.*
 - *Especificar nome.*
 - *Especificar identificação do usuário.*
 - *Especificar PIN e senha.*
- *Especificar número da receita.*
- *Especificar se é exigida a data de revalidação.*

Para completar a tarefa, são definidas três subtarefas. Uma das subtarefas, *fornecer informações de identificação*, é elaborada ainda mais em três subsubtarefas adicionais.

11.3.3 Análise do conteúdo exibido

As tarefas de usuário identificadas na Seção 11.3.2 levam à apresentação de uma série de tipos diferentes de conteúdo. Para aplicações modernas, o conteúdo exibido pode variar de relatórios de texto (por exemplo, uma planilha), visualização gráfica (por exemplo, um histograma, um modelo 3-D, a foto de uma pessoa) ou informações especializadas (por exemplo, arquivos de áudio ou vídeo). As técnicas de modelagem de análise discutidas nos Capítulos 6 e 7 identificam os objetos de dados de saída produzidos por uma aplicação. Tais objetos de dados poderiam ser: (1) gerados por componentes (não relacionados com a interface) em outras partes de uma aplicação, (2) obtidos de dados armazenados em um banco de dados acessível para a aplicação ou (3) transmitidos de sistemas externos à aplicação em questão.

Durante essa etapa de análise de interface, são considerados o formato e a estética do conteúdo (como ele é exibido pela interface). Entre as perguntas feitas e respondidas, temos:

- Os diferentes tipos de dados são alocados em posições geográficas consistentes na tela (por exemplo, fotos sempre apareceriam no canto superior direito)?
- O usuário pode personalizar a localização do conteúdo na tela?
- Foi atribuída identificação apropriada na tela para todos os conteúdos?
- Se for preciso apresentar um relatório grande, como seria subdividido para facilitar sua compreensão?
- Haverá mecanismos disponíveis para ir diretamente a informações resumidas em conjuntos de dados volumosos?

 Como determinar o formato e a estética de conteúdo exibido como parte da interface do usuário?

- A saída gráfica será apresentada em escala para caber nos limites do dispositivo de exibição utilizado?
- Como serão usadas cores para melhorar o entendimento?
- Como serão apresentadas ao usuário mensagens de erro e alertas?

As respostas a essas (e outras) questões nos ajudarão a estabelecer os requisitos para apresentação de conteúdo.

11.3.4 Análise do ambiente de trabalho

Hackos e Redish [Hac98] discutem a importância da análise do ambiente de trabalho ao afirmarem:

As pessoas não realizam seus trabalhos de forma isolada. São influenciadas pela atividade em torno delas, pelas características físicas do local de trabalho, pelo tipo de equipamento utilizado e pelas relações de trabalho que têm com outras pessoas. Se os produtos projetados não se ajustarem ao ambiente, serão difíceis ou frustrantes de ser usados.

Em algumas aplicações, a interface do usuário para um sistema baseado em computador é colocada em uma “posição que facilita o usuário” (por exemplo, iluminação apropriada, altura adequada da tela, fácil acesso ao teclado), porém em outras (por exemplo, no chão de fábrica ou no *cockpit* de um avião), talvez a iluminação não seja tão adequada, o ruído pode ser um fator importante, um teclado ou mouse talvez não sejam uma opção, o posicionamento da tela talvez seja abaixo do ideal. O projetista de interfaces talvez esteja restrito por fatores que reduzam a facilidade de uso.

Além dos fatores ambientais físicos, a cultura do local de trabalho também entra em cena. A interação do sistema será medida de alguma maneira (por exemplo, tempo por transação ou precisão de uma transação)? Duas ou mais pessoas terão de compartilhar informações antes de uma opinião poder ser fornecida? Como será oferecido suporte aos usuários do sistema? Essas e muitas outras questões relacionadas devem ser respondidas antes de o projeto de interface se iniciar.

11.4 ETAPAS NO PROJETO DE INTERFACES

“O projeto interativo [é] uma mescla perfeita de arte gráfica, tecnologia e psicologia.”

Brad Wieners

Assim que a análise de interface tiver sido completada, todas as tarefas (ou objetos e ações) requeridas pelo usuário final foram identificadas de forma detalhada e a atividade de projeto de interface se inicia. O projeto de interfaces, assim como todo projeto de engenharia de software, é um processo iterativo. Cada etapa de um projeto de interface do usuário ocorre uma série de vezes, elaborando e refinando informações desenvolvidas na etapa anterior.

Embora tenham sido propostos diversos modelos diferentes para projeto de interfaces do usuário (por exemplo, [Nor86], [Nie00]), todos sugerem alguma combinação das seguintes etapas:

1. Usar informações desenvolvidas durante a análise de interfaces (Seção 11.3), definir objetos e ações (operações) de interface.
2. Definir eventos (ações de usuário) que provocarão a mudança de estado de uma interface do usuário. Modelar esse comportamento.
3. Representar cada estado da interface como realmente aparecerá para o usuário final.
4. Indicar como o usuário interpreta o estado do sistema com base em informações fornecidas através da interface.

Em alguns casos, podemos começar com esboços de cada estado de interface (ou seja, qual o aspecto da interface do usuário sob diversas circunstâncias) e então trabalhar no sentido inverso para definir objetos, ações e outras importantes informações de projeto. Independentemente da sequência de tarefas de projeto, devemos: (1) sempre seguir as regras de ouro discutidas na

Seção 11.1, (2) modelar como a interface será implementada e (3) considerar o ambiente (por exemplo, tecnologia de exibição, sistema operacional, ferramentas de desenvolvimento) a ser usado.

11.4.1 Aplicação das etapas para projeto de interfaces

A definição dos objetos da interface e as ações aplicadas a eles é uma importante etapa no projeto. Para concretizar isso, cenários de usuário são analisados sintaticamente quase da mesma forma descrita no Capítulo 6. Ou seja, é escrito um caso de uso. Os substantivos (objetos) e os verbos (ações) são isolados para criar uma lista de objetos e ações.

Assim que os objetos e ações tiverem sido definidos e elaborados iterativamente, são classificados por tipo. Identificam-se objetos de destino, origem e aplicação. Um *objeto de origem* (por exemplo, um ícone de relatório) é arrastado e solto sobre um *objeto de destino* (por exemplo, um ícone de impressora). A implicação dessa ação é criar um relatório impresso. Um *objeto de aplicação* representa dados específicos à aplicação que não são diretamente manipulados como parte da interação de tela. Por exemplo, uma lista de endereços é usada para armazenar nomes para uma postagem. A própria lista poderia ser classificada, combinada ou filtrada (ações baseadas em menus), mas ela não é arrastada e solta através de interação com o usuário.

Quando estiver satisfeito com todas as ações e objetos importantes definidos (para uma iteração de projeto), realize o layout da tela. Assim como outras atividades do projeto de interfaces, o layout da tela é um processo interativo no qual são realizados o projeto gráfico e o posicionamento dos ícones, a definição de texto de tela descritivo, a especificação e a colocação de títulos para as janelas, bem como a definição de itens de menu principais e secundários. Se for apropriada para a aplicação uma metáfora do mundo real, esta é especificada neste momento e o layout é organizado para complementar tal metáfora.

Para darmos um breve exemplo das etapas de projeto citadas anteriormente, consideremos um cenário de usuário para o sistema *CasaSegura* (discutido em capítulos anteriores). Segue um caso de uso preliminar (redigido pelo proprietário do imóvel) para a interface:

Caso de uso preliminar: Quero ter acesso ao meu sistema *CasaSegura* de qualquer ponto remoto via Internet. Por meio de um navegador instalado em meu notebook (enquanto estou no trabalho ou viajando), posso determinar o estado do sistema de alarme, armar ou desarmá-lo, reconfigurar zonas de segurança e ver diferentes ambientes da casa via câmeras de vídeo pré-instaladas.

Para acessar o *CasaSegura* de um ponto remoto, forneço um identificador de usuário e uma senha. Estes definem níveis de acesso (por exemplo, nem todo usuário poderá reconfigurar o sistema) e dão segurança. Uma vez validados, posso verificar o estado do sistema e alterá-lo armando ou desarmando o *CasaSegura*. Posso reconfigurar o sistema exibindo uma planta da casa, vendo cada um dos sensores de segurança, exibindo cada zona configurada atualmente e modificando as zonas conforme necessário. Posso ver o interior da casa via câmeras de vídeo estrategicamente colocadas. Posso deslocar e ampliar a imagem de cada câmera para ter visões diferentes do seu interior.

Tomando como base esse caso de uso, são identificados as seguintes tarefas, objetos e dados do proprietário do imóvel:

- *Acessa* o sistema *CasaSegura*.
- *Introduz* um **ID** e **senha** para permitir acesso remoto.
- *Verifica* **estado do sistema**.
- *Arma* ou *desarma* o sistema *CasaSegura*.
- *Exibe* **planta** e **localização dos sensores**.
- *Exibe* **zonas** na planta.
- *Altera* **zonas** na planta.
- *Exibe* **localização das câmeras de vídeo** na planta.
- *Seleciona* **câmera de vídeo** para visualização.
- *Visualiza* **imagens de vídeo** (quatro quadros por segundo).
- *Desloca* ou *amplia* o **foco da câmera de vídeo**.



Embora as ferramentas automatizadas possam ser úteis no desenvolvimento de protótipos de layout, algumas vezes basta um lápis e papel.

Os objetos (em negrito) e as ações (em *itálico*) são extraídos da lista de tarefas do proprietário do imóvel. A maioria dos objetos citados são objetos de aplicação. Entretanto, **localização das câmeras de vídeo** (um objeto de origem) é arrastado e solto sobre **câmera de vídeo** (um objeto de destino) para criar uma **imagem de vídeo** (uma janela com exibição de vídeo).

É criado um esboço preliminar do layout da tela para monitoramento de vídeo (Figura 11.3).⁶ Para chamar a imagem de vídeo, é selecionado um ícone de localização de câmera de vídeo, *C*, localizado na planta exibida na janela de monitoramento. Nesse caso, a localização da câmera na sala (SE) é então arrastada e solta sobre o ícone de câmera de vídeo no canto superior esquerdo da tela. Surge a janela de imagem de vídeo, mostrando vídeo *streaming* da câmera localizada em SE. Os controles deslizantes para controle de ampliação e deslocamento são usados para controlar a ampliação e a direção da imagem de vídeo. Para selecionar uma vista de outra câmera, o usuário apenas arrasta e solta um ícone de localização da câmera diferente sobre o ícone de câmera no canto superior esquerdo da tela.

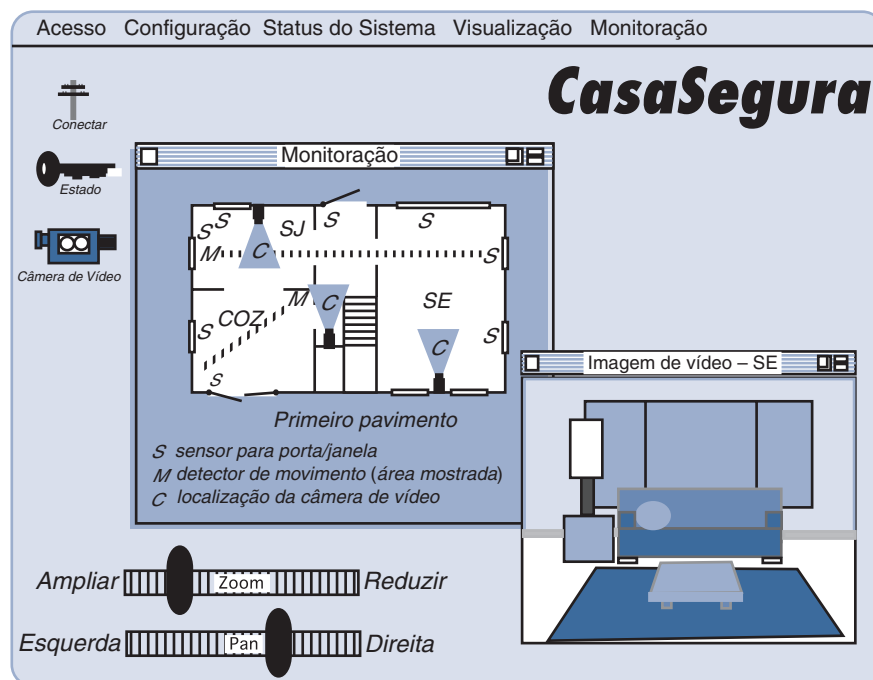
O esboço do layout teria de ser complementado com uma expansão de cada item de menu contido na barra de menus, indicando que ações estão disponíveis para o modo (estado) de monitoramento de vídeo. Um conjunto completo de esboços para cada tarefa do proprietário do imóvel citada no cenário de usuário seria criado durante o projeto de interfaces.

11.4.2 Padrões de projeto de interfaces do usuário

As interfaces gráficas do usuário tornaram-se tão comuns que surgiu uma ampla gama de padrões de projeto de interfaces. Conforme citado anteriormente neste livro, um padrão de projeto é uma abstração que prescreve uma solução de projeto para um problema de projeto específico e bem delimitado.

FIGURA 11.3

Layout preliminar da tela



⁶ Observe que este difere ligeiramente da implementação desses recursos em capítulos anteriores. Isso poderia ser considerado um projeto preliminar e representa uma alternativa que possa vir a ser considerada.

WebRef

Foi proposta uma ampla gama de padrões de projeto para interfaces do usuário. Para encontrar uma grande variedade de sites com esse tipo de padrões, visite www.hcipatterns.org.

Como exemplo de problema comumente encontrado para projeto de interfaces, consideremos a situação em que um usuário tem de introduzir uma ou mais datas, às vezes com meses de antecedência. Existem várias soluções possíveis para esse problema simples e uma série de padrões diferentes que poderiam ser propostos. Laakso [Laa00] sugere um padrão denominado **FaixaDeCalendário**, que produz um calendário contínuo e que rola onde a data atual é destacada e datas futuras podem ser selecionadas indicando-as no calendário. A metáfora de calendário é bem conhecida de todo usuário e oferece um mecanismo efetivo para colocar uma data futura no contexto.

Ao longo da última década foram propostos vários padrões de projeto de interfaces. Uma discussão mais detalhada sobre padrões de projeto de interfaces do usuário é apresentada no Capítulo 12. Além disso, Erickson [Eri08] indica links para vários conjuntos baseados na Web.

11.4.3 Questões de projeto

À medida que o projeto de uma interface do usuário evolui, quatro questões de projeto comuns quase sempre vêm à tona: tempo de resposta do sistema, recursos de ajuda ao usuário, informações de tratamento de erros e atribuição de nomes a comandos. Infelizmente, muitos projetistas não tratam desses problemas até um ponto relativamente avançado do processo de projeto (algumas vezes a primeira vaga ideia de um problema não ocorre até que um protótipo operacional esteja disponível). Em geral, decorrem iterações desnecessárias, atrasos de projeto e frustração por parte do usuário final. É muito melhor estabelecer cada um desses como um problema de projeto a ser considerado no início do projeto de software, quando as mudanças são fáceis e custam pouco.

Tempo de resposta. O tempo de resposta do sistema é a principal queixa para várias aplicações interativas. Em geral, o tempo de resposta do sistema vai do ponto no qual o usuário realiza alguma ação de controle (por exemplo, aciona a tecla <Enter> ou clica o mouse) até que o software responda com saída ou ação desejadas.

O tempo de resposta do sistema apresenta duas importantes características: duração e variabilidade. Se a resposta do sistema for muito longa, frustração e estresse por parte do usuário serão inevitáveis. A *variabilidade* refere-se ao desvio do tempo de resposta médio e, em vários aspectos, é a característica de tempo de resposta mais importante. Baixa variabilidade permite ao usuário estabelecer um ritmo de interação, mesmo que o tempo de resposta seja relativamente longo. Por exemplo, uma resposta de 1 segundo a um comando normalmente é preferível do que uma resposta que varia de 0,1 a 2,5 segundos. Quando a variabilidade é significativa, o usuário sempre se desequilibra, sempre conjecturando se algo “diferente” ocorreu ou não nos bastidores.

Recursos de ajuda. Quase todo usuário de um sistema computacional interativo requer ajuda de vez em quando. Em alguns casos, uma simples questão dirigida a um colega com bons conhecimentos pode resolver a questão. Em outros, uma pesquisa detalhada em um conjunto de vários volumes de “manuais de usuário” talvez seja a única alternativa. Na maioria das vezes, entretanto, os aplicativos modernos fornecem recursos de ajuda on-line que permitem a um usuário obter a resposta para determinada questão ou resolver um problema sem ter de abandonar a interface.

Uma série de problemas de projeto [Rub88] tem de ser tratada quando se considera um recurso de ajuda de sistema:

- A ajuda estará disponível para todas as funções do sistema e a qualquer momento durante a interação com o sistema? Há a possibilidade de incluir ajuda apenas para um subconjunto de todas as funções e ações ou ajuda para todas as funções.
- Como o usuário solicitará ajuda? Entre as opções temos um menu de ajuda, uma tecla de função especial ou um comando HELP.
- Como será representada a ajuda? Entre as opções temos uma janela separada, uma referência a um documento impresso (não ideal) ou uma sugestão de uma ou duas linhas produzida em uma localização fixa da tela.

“Um erro comum que as pessoas cometem ao tentar projetar algo completamente infalível é subestimar a criatividade de verdadeiros tolos.”

Douglas Adams

- Como o usuário irá retornar à interação normal? Entre as opções temos um botão de retorno exibido na tela, uma tecla de função ou uma sequência de controle.
- Como as informações do sistema de ajuda serão estruturadas? Entre as opções temos uma estrutura “plana” em que todas as informações são acessadas por meio de uma palavra-chave, uma hierarquia de informações em camadas que dá cada vez mais detalhes à medida que o usuário vai avançando nessa estrutura ou o emprego de hipertexto.

“A interface do inferno: ‘Para corrigir esse erro e prosseguir, introduza qualquer número primo de 11 dígitos ...’”

Autor desconhecido

Tratamento de erros. As mensagens de erros e alertas são “más notícias” fornecidas aos usuários de sistemas interativos quando algo saiu mal. No pior caso, as mensagens de erro e alertas transmitem informações inúteis ou confusas e servem apenas para aumentar a frustração do usuário. Há poucos usuários de computador que ainda não depararam com um erro do tipo: *‘A aplicação XXX foi forçada a sair; pois um erro do tipo 1023 foi encontrado’*. Em algum lugar deve existir uma explicação para o erro 1023; caso contrário, para que aqueles que projetaram o sistema colocaram tal identificação? Contudo, a mensagem de erro não dá nenhuma indicação real do que deu errado ou onde procurar informações adicionais. Uma mensagem de erro apresentada dessa maneira não faz nada para diminuir a ansiedade do usuário ou ajudá-lo a corrigir o problema.

Em geral, toda mensagem de erro ou alerta produzida por um sistema interativo deve apresentar as seguintes características:

- A mensagem deve descrever o problema em um jargão que o usuário consiga entender.
- A mensagem deve fornecer conselhos construtivos para recuperação do erro.
- A mensagem deve indicar quaisquer consequências negativas do erro (por exemplo, arquivos de dados provavelmente corrompidos), de modo que o usuário possa fazer uma verificação para garantir que não tenham ocorrido (ou corrigi-las, caso tenham ocorrido).
- A mensagem deve ser acompanhada por algum sinal audível ou visual. Ou seja, poderia ser gerado um bipe acompanhando a exibição da mensagem ou a mensagem poderia piscar momentaneamente ou ser exibida em uma cor facilmente reconhecível como “cor de erro”.
- A mensagem deve ser “não sentenciosa”. Isto é, os termos jamais devem colocar a culpa no usuário.

Pelo fato de ninguém gostar de más notícias, poucos usuários apreciarão uma mensagem de erro independentemente da qualidade de sua formulação. Porém, uma filosofia de mensagens de erro efetiva pode contribuir muito para aumentar a qualidade de um sistema interativo e reduzirá significativamente a frustração dos usuários quando da ocorrência de problemas.

Atribuição de nomes a comandos e menus. O comando digitado já foi o modo mais comum de interação entre o usuário e um software de sistema e era usado comumente para aplicações de todo tipo. Hoje em dia, o uso de interfaces orientadas a janelas e apontar e clicar reduziram a dependência de comandos digitados, mas alguns usuários com maior conhecimento ainda preferem um modo de interação orientado a comandos. Surge uma série de problemas de projeto quando são fornecidos comandos digitados ou identificadores de menus como modo de interação:

- Toda opção de menu terá um comando correspondente?
- Qual a forma a ser assumida pelos comandos? As opções incluem uma sequência de controle (por exemplo, alt-P), teclas de função ou uma palavra digitada.
- Qual será o grau de dificuldade para aprender e lembrar-se dos comandos? O que pode ser feito se um comando for esquecido?
- Os comandos podem ser personalizados ou abreviados pelo usuário?
- Os identificadores de menus são autoexplicativos no contexto da interface?
- Os submenus são consistentes com a função sugerida por um item de menu principal?

Quais características uma “boa” mensagem de erro deve apresentar?

Conforme citado neste capítulo, as convenções para o uso de comandos devem ser estabelecidas para todas as aplicações. É confuso e normalmente sujeito a erros um usuário digitar alt-D quando um objeto gráfico tem de ser duplicado em determinada aplicação e alt-D quando um objeto gráfico tem de ser deletado em outra aplicação. A probabilidade para ocorrência de erro é óbvia.

WebRef

Diretrizes para o desenvolvimento de software acessível podem ser encontradas em www3.ibm.com/able/guidelines/software/accesssoftware.html.

Acessibilidade às aplicações. À medida que as aplicações de computador tornam-se comuns, os engenheiros de software devem garantir que o projeto de interfaces englobe mecanismos que permitam fácil acesso para aqueles com necessidades especiais. A *acessibilidade* para usuários (e engenheiros de software) que podem vir a ser desafiados fisicamente é um imperativo por razões éticas, legais e comerciais. Uma grande variedade de diretrizes de acessibilidade (por exemplo, [W3C03]) — muitas projetadas para aplicações Web, mas em geral aplicáveis a todos os tipos de software — dão sugestões detalhadas para projeto de interfaces que atingem níveis de acessibilidade variados. Outros (por exemplo, [App08], [Mic08]) apresentam orientações específicas para “tecnologia assistente” que lida com as necessidades daqueles com problemas visuais, auditivos, motores, de fala e de aprendizado.

Internacionalização. Os engenheiros de software e seus gerentes invariavelmente subestimam o esforço e as capacidades necessárias para criar interfaces do usuário que levem em conta as necessidades de diferentes localidades e idiomas. Muitas vezes, as interfaces são desenhadas para um país e idioma e então improvisadas para funcionar em outros. O desafio para os projetistas de interfaces é criar um software “globalizado”. Isto é, as interfaces com o usuário devem ser projetadas para atender um núcleo genérico de funcionalidade que possa ser entregue a todos que usam o software. Recursos de *localização* permitem que a interface seja personalizada para um mercado específico.

Uma grande variedade de diretrizes de internacionalização (por exemplo, [IBM03]) se encontra disponível para os engenheiros de software. Tais diretrizes tratam de ampla gama de questões de projeto (por exemplo, os layouts de tela talvez difiram em vários mercados) e problemas de implementação diferenciados (por exemplo, diferentes alfabetos talvez criem exigências especiais de atribuição de nomes e espaçamento). O padrão *Unicode* [Uni03] foi desenvolvido para tratar do intimidante desafio de gerenciar dezenas de linguagens naturais com centenas de caracteres e símbolos.

FERRAMENTAS DO SOFTWARE



Desenvolvimento de interfaces do usuário

Objetivo: Essas ferramentas permitem a um engenheiro de software criar uma sofisticada interface gráfica do usuário com relativamente pouco desenvolvimento de software personalizado. As ferramentas dão acesso a componentes reutilizáveis e tornam a criação da interface uma simples questão de selecionar capacidades predefinidas que são agrupadas usando-se a ferramenta.

Mecânica: As interfaces do usuário modernas são construídas usando-se um conjunto de componentes reutilizáveis associados a alguns componentes personalizados desenvolvidos para oferecer recursos especializados. A maioria das ferramentas para desenvolvimento de interfaces com o usuário permite que um engenheiro de software crie uma interface usando recursos de “arrastar e soltar”. Isto é, o desenvolvedor escolhe alguns de vários recursos predefinidos (por exemplo, recursos de construção de formulários, mecanismos de interação, processamento

de comandos) e insere tais recursos no conteúdo da interface a ser criada.

Ferramentas representativas:⁷

LegaSuite GUI, desenvolvida pela Seagull Software (www.seagullsoftware.com), permite a criação de GUIs baseadas em navegadores e oferece recursos para a reengenharia de interfaces antiquadas.

Motif Common Desktop Environment, desenvolvida pelo The Open Group (www.osf.org/tech/desktop/cde/), é uma interface gráfica do usuário integrada para sistemas desktop abertos. Ela oferece uma única interface gráfica padrão para o gerenciamento de dados e arquivos (a área de trabalho gráfica) e aplicações.

Altia Design 8.0, desenvolvida pela Altia (www.altia.com), é uma ferramenta para criação de GUIs em uma variedade de plataformas diferentes (por exemplo, automotiva, dispositivos portáteis, industrial).

⁷ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria.

11.5 PROJETO DE INTERFACES PARA WEBAPP

Toda interface do usuário — seja ela desenhada para uma WebApp, uma aplicação de software tradicional, um produto de consumo ou para um dispositivo industrial — deve apresentar as características de usabilidade discutidas neste capítulo. Dix [Dix99] argumenta que se deve projetar uma interface para WebApp de modo que responda a três perguntas primárias para o usuário final:



Se for provável que os usuários entrarão em sua WebApp em vários pontos e níveis dentro da hierarquia de conteúdo, certifique-se de desenhar cada página com recursos de navegação que levem o usuário a outros pontos de interesse.

Onde me encontro? A interface deve: (1) dar uma indicação da WebApp que foi acessada⁸ e (2) informar o usuário a sua localização na hierarquia de conteúdos.

O que posso fazer agora? A interface sempre deve auxiliar o usuário a entender suas opções atuais — que funções estão disponíveis, quais links ainda existem e qual conteúdo é relevante?

Onde estive, para onde estou indo? A interface deve facilitar a navegação. Portanto, tem de fornecer um “mapa” (implementado de maneira fácil de ser compreendido) de onde o usuário esteve e quais caminhos devem ser tomados para ir a algum outro ponto dentro da WebApp.

Uma interface WebApp eficaz deve dar respostas a cada uma dessas questões enquanto o usuário final navega pelo conteúdo e funcionalidade.

11.5.1 Princípios e diretrizes para projeto de interfaces

A interface do usuário de uma WebApp é sua “primeira impressão”. Independentemente do valor de seu conteúdo, da sofisticação de seus recursos e serviços de processamento, bem como do benefício geral da própria WebApp, uma interface malfeita desapontará o usuário potencial e talvez faça com que, de fato, procure outra opção. Devido ao grande número de WebApps concorrentes em praticamente qualquer área de aplicação, a interface tem de “atrair” imediatamente um possível usuário.

Bruce Tognozzi [Tog01] define um conjunto de características fundamentais que todas as interfaces deveriam apresentar e, ao fazer isso, estabelece uma filosofia que deveria ser seguida por todo projetista de interfaces para WebApps:

Interfaces eficazes são visualmente aparentes e “generosas”, instilando em seus usuários uma sensação de controle. Os usuários veem rapidamente o leque de opções, captam como atingir seus objetivos e realizar seus trabalhos.

Interfaces eficazes não fazem com que o usuário se preocupe com os detalhes de funcionamento interno do sistema. O trabalho é cuidadosa e continuamente salvo, com total possibilidade de o usuário desfazer qualquer atividade a qualquer momento.

Aplicações e serviços eficazes realizam o máximo de trabalho, exigindo o mínimo de informações dos usuários.

Para poder projetar interfaces para WebApp que apresentem essas características, Tognozzi [Tog01] identifica um conjunto de princípios de projeto⁹ primordiais:

Antecipação. *Uma WebApp deve ser desenhada para prever o próximo passo do usuário.* Consideremos, por exemplo, uma WebApp de apoio ao cliente desenvolvida por um fabricante de impressoras. Um usuário solicitou um objeto de conteúdo que apresente informações sobre um driver de impressora para um sistema operacional recém-lançado. O projetista da WebApp deve prever que o usuário possa vir a solicitar um download do driver e deve oferecer recursos de navegação que permitam que isso aconteça sem exigir do usuário a busca desse recurso.

“Se um site for perfeitamente utilizável, mas lhe faltar um estilo de projeto elegante e apropriado, ele estará fadado ao fracasso.”

Curt Cloninger

PONTO-CHAVE

Uma interface para WebApp de qualidade deve ser compreensível e flexível, dando ao usuário a sensação de controle.

Existe um conjunto de princípios básicos que possam ser aplicados à medida que desenhemos uma GUI?

⁸ Cada um de nós já passou pela experiência de criar um bookmark de uma página Web e ao revisita-la não ter indicação alguma do site ou do contexto para tal página Web (bem como nenhuma forma de se deslocar para outro local no site).

⁹ Os princípios de Tognozzi originais foram adaptados e estendidos para uso neste livro. Veja [Tog01] para uma discussão mais ampla sobre esses princípios.

Comunicação. *A interface deve comunicar o estado de qualquer atividade iniciada pelo usuário. A comunicação deve ser óbvia (por exemplo, uma mensagem de texto) ou sutil (por exemplo, a imagem de uma folha de papel deslocando-se pela impressora para indicar que a impressão está em andamento). A interface também deve comunicar o estado do usuário (por exemplo, a identificação do usuário) e sua localização na hierarquia de conteúdos da WebApp.*

Consistência. *O uso de controles de navegação, menus, ícones e estética (por exemplo, cor, forma, layout) devem ser consistentes em toda a WebApp. Por exemplo, se texto sublinhado na cor azul sugerir um link de navegação, o conteúdo jamais deveria incorporar texto sublinhado azul que não indique um link. Além disso, um objeto, digamos, um triângulo amarelo, usado para indicar uma mensagem de alerta antes de o usuário chamar determinada função ou ação, não deve ser usado para outras finalidades em nenhum outro ponto da WebApp. Por fim, todo recurso da interface deve responder de maneira consistente com as expectativas¹⁰ do usuário.*

Autonomia controlada. *A interface deve facilitar a movimentação do usuário pela WebApp, mas deve fazê-lo de forma que faça valer convenções de navegação estabelecidas para a aplicação. Por exemplo, a navegação em trechos de segurança da WebApp deve ser controlada por meio de identificação e senhas de usuário e não deve existir nenhum mecanismo de navegação que possibilite a um usuário evitar tais controles.*

Eficiência. *O projeto de uma WebApp e sua interface devem otimizar a eficiência de trabalho do usuário, e não a eficiência do desenvolvedor que a projeta e constrói ou o ambiente cliente/servidor que a executa. Tognozzi [Tog01] discute isso ao escrever: “Esta simples verdade é a razão de ser tão importante para todos os envolvidos em um projeto de software perceber a importância de fazer com que a produtividade do usuário seja a primeira meta e compreender a diferença vital entre construir um sistema eficiente e dar poderes a um usuário eficiente”.*

Flexibilidade. *A interface deve ser flexível o bastante para permitir que alguns usuários cumpram tarefas diretamente ao passo que outros devam explorar a WebApp de maneira um tanto aleatória. Em todos os casos, ela deve permitir ao usuário compreender onde ele se encontra e também lhe dar um recurso para desfazer erros e refazer caminhos de navegação mal escolhidos.*

Foco. *A interface para WebApp (e o conteúdo por ela apresentado) devem permanecer focados na(s) tarefa(s) do usuário em questão. Em toda hipermídia há uma tendência de direcionar o usuário a conteúdo de pouca relação. Por quê? Porque é muito fácil fazê-lo! O problema é que o usuário pode rapidamente se perder em muitas camadas de informações de apoio e perder de vista o conteúdo original que desejava de início.*

Lei de Fitt. *“O tempo para atingir um alvo é função da distância e do tamanho do alvo” [Tog01]. Baseado em um estudo realizado nos anos 1950 [Fit54], a lei de Fitt “é um método eficaz de modelar movimentos rápidos e dirigidos, em que um apêndice (como uma mão), inicia em repouso em uma posição inicial específica e se desloca para o repouso em uma área-alvo” [Zha02]. Se a sequência de seleções ou entradas padronizadas (com muitas opções diferentes na sequência) for definida por uma tarefa e usuário, a primeira seleção (por exemplo, uma seleção com o mouse) deve ser fisicamente próxima da seleção seguinte. Consideremos, por exemplo, uma interface de homepage de uma WebApp em um site de comércio eletrônico que vende produtos eletrônicos de consumo.*

Cada opção de usuário implica um conjunto de escolhas ou ações de usuário que vêm em seguida. Por exemplo, a opção “comprar um produto” requer que o usuário introduza uma categoria de produto seguida pelo nome do produto. A categoria do produto (por exemplo, equipamento de áudio, televisores, aparelhos de DVD) aparece como um menu pull-down assim que a opção “comprar um produto” for selecionada. Consequentemente, a escolha seguinte é óbvia à primeira vista (ela se encontra nas proximidades), e o tempo para reconhecê-la será desprezível.

“O melhor caminho é aquele com o menor número de passos. Diminua a distância entre o usuário e sua meta.”

Autor desconhecido

WebRef

Uma busca na Web irá revelar diversas bibliotecas disponíveis como, por exemplo, pacotes de API Java, interfaces e classes, em java.sun.com ou COM, DCOM e Type Libraries em msdn.microsoft.com.

¹⁰ Tognozzi [Tog01] observa que a única maneira de garantir que as expectativas de um usuário sejam adequadamente compreendidas é por meio de exaustivos testes com o usuário (Capítulo 20).

Se, por outro lado, a escolha aparecesse em um menu que estivesse localizado do outro lado da tela, o tempo para o usuário reconhecê-lo (e depois fazer sua escolha) seria bem mais longo.

Objeto de interface humana. *Desenvolveu-se uma vasta biblioteca de objetos de interface humana reutilizáveis para WebApps. Use-a.* Qualquer objeto de interface que possa ser “visto, ouvido, tocado ou de alguma outra forma percebido” [Tog01] por um usuário final pode ser obtido de qualquer uma das várias bibliotecas de objetos.

Redução da latência. *Em vez de fazer com que o usuário espere por alguma operação interna completar (por exemplo, baixar uma imagem complexa), a WebApp deve usar multitarefas de uma forma que deixe o usuário prosseguir com seu trabalho como se a operação tivesse sido completada.* Além de reduzir a latência, devem ser reconhecidos atrasos, de modo que o usuário entenda o que está ocorrendo. Isso inclui: (1) fornecer feedback de áudio quando uma seleção não resulte em uma ação imediata pela WebApp, (2) exibir uma animação de relógio ou barra de progresso para indicar que o processamento está em andamento e (3) fornecer algum entretenimento (por exemplo, uma apresentação de texto ou animação) enquanto ocorre processamento moroso.

Facilidade de aprendizagem. *Uma interface para WebApp deve ser projetada para minimizar o tempo de aprendizagem e, uma vez aprendida, minimizar a reaprendizagem necessária quando a WebApp for reutilizada.* Em geral, a interface deve enfatizar um projeto simples e intuitivo que organiza conteúdo e funcionalidade em categorias que são óbvias para o usuário.



Metáforas são uma excelente ideia, pois espelham a experiência do mundo real. Apenas certifique-se de que a metáfora escolhida seja bem conhecida dos usuários finais.

Metáforas. *Uma interface que usa uma metáfora de interação é mais fácil de aprender e usar, desde que a metáfora seja apropriada para a aplicação e o usuário.* A metáfora deve invocar imagens e conceitos da experiência do usuário, mas não precisa ser uma reprodução exata de uma experiência do mundo real. Por exemplo, um site de comércio eletrônico que implemente pagamento automatizado de contas para uma instituição financeira, usa uma metáfora talonário de cheques (não é de surpreender) para auxiliar o usuário a especificar e programar pagamentos de contas. Entretanto, quando um usuário “preenche” um cheque, ele não precisa introduzir o nome completo do beneficiário, mas poderá sim escolher de uma lista de beneficiários ou fazer com que o sistema o selecione baseado logo nas primeiras letras digitadas. A metáfora permanece intacta, porém o usuário obtém uma ajuda da WebApp.

Manter a integridade do artefato. *Um artefato (por exemplo, um formulário preenchido pelo usuário, uma lista especificada pelo usuário) deve ser salvo automaticamente para não ser perdido caso ocorra algum erro.* Todos já experimentamos a frustração associada a completar um enorme formulário de WebApp apenas para ter o conteúdo perdido devido a um erro (por nós cometido, cometido pela WebApp ou durante a transmissão do cliente para o servidor). Para evitar isso, uma WebApp deve ser projetada para salvar automaticamente todos os dados especificados pelo usuário. A interface deve oferecer suporte a essa função e oferecer ao usuário um mecanismo fácil para recuperar informações “perdidas”.

Legibilidade. *Todas as informações apresentadas em uma interface devem ser legíveis por jovens e idosos.* O projetista de interfaces deve dar prioridade a estilos de tipos e tamanhos de fontes legíveis e fundos coloridos que aumentem o contraste.

Acompanhar o estado de interação. *Quando apropriado, o estado da interação de usuário deve ser acompanhado e armazenado de modo que um usuário possa sair do sistema e retornar mais tarde, prosseguindo do ponto onde parou.* Em geral, podem ser projetados cookies para armazenar informações de estado. Entretanto, os cookies são uma tecnologia controversa e outras soluções de projeto talvez sejam mais aceitáveis para alguns usuários.

Navegação visível. *Uma interface para WebApp bem projetada fornece “a ilusão de que os usuários se encontram no mesmo lugar, com o trabalho sendo levado a eles” [Tog01].* Quando é usada esta abordagem, a navegação não é uma preocupação para o usuário. Ao contrário, o usuário recupera objetos de conteúdo e seleciona funções que são exibidas e executadas através da interface.

CASASEGURA

**Revisão de projeto de interfaces**

Cena: Sala de Doug Miller.

Atores: Doug Miller (gerente do grupo de engenharia de software do CasaSegura) e Vinod Raman, membro da equipe de engenharia de software do produto CasaSegura.

Conversa:

Doug: Vinod, você e a equipe tiveram a chance de revisar o protótipo de interface para comércio eletrônico **CasaSegura-Garantida.com**?

Vinod: Sim... Todos nós navegamos nele de um ponto de vista técnico e tenho um bocado de comentários. Ontem, os enviei por e-mail a Sharon [gerente da equipe de WebApps do fornecedor terceirizado para o site de comércio eletrônico CasaSegura].

Doug: Você e a Sharon podiam se reunir e discutir os pequenos detalhes... Entregue-me um resumo das questões importantes.

Vinod: Em termos gerais, eles fizeram um bom trabalho, nada de revolucionário, porém é uma interface típica para comércio eletrônico, estética aceitável, layout razoável, cobriram todas as funções importantes...

Doug (sorrindo tristemente): Mas?

Vinod: Bem, há algumas coisinhas...

Doug: Como... Vinod (**mostrando a Doug a sequência de storyboards para o protótipo de interface**): Aqui está o menu das funções principais que é apresentado na homepage:

Conheça o CasaSegura.

Descreva sua casa.

Obtenha recomendações de componentes para o CasaSegura.

Adquira um sistema CasaSegura.

Solicite suporte técnico.

O problema não é com essas funções. Todas estão corretas, mas o nível de abstração não.

Doug: Todas são funções importantes, não é mesmo?

Vinod: Sim, são, mas aí é que está o problema... É possível comprar um sistema apenas entrando com uma lista de componentes... Não há necessidade de descrever a casa se não quiser fazê-lo. Sugeriria apenas quatro opções de menu na homepage:

Conheça o CasaSegura.

Especifique o sistema CasaSegura que você precisa.

Adquira um sistema CasaSegura.

Solicite suporte técnico.

Ao selecionar **Especifique o sistema CasaSegura que você precisa**, você terá então as seguintes opções:

Selecione os componentes para o CasaSegura.

Obtenha recomendações de componentes para o CasaSegura.

Se você for um usuário experiente, escolherá componentes de um conjunto de menus pull-down classificados por sensores, câmeras, painéis de controle etc. Se precisar de ajuda, você solicitará uma recomendação, que exigirá que descreva sua casa. Imagino que seja um pouco mais lógico.

Doug: Concordo. Você conversou com a Sharon a esse respeito?

Vinod: Não, queria discutir isso primeiro com o Marketing; depois telefonaria para ela.

Nielsen e Wagner [Nie96] sugerem algumas diretrizes pragmáticas para projeto de interfaces (baseadas na experiência dos autores no reprojeção de uma importante WebApp) que são um excelente complemento aos princípios sugeridos anteriormente nesta seção:

"As pessoas têm muito pouca paciência com sites WWW mal projetados."

Jakob Nielsen e Annette Wagner

- Ler rápido em um monitor de computador é aproximadamente 25% mais lento que ler rápido em papel. Consequentemente, não force o usuário a ler toneladas de texto, particularmente quando o texto explica a operação da WebApp ou auxilia na navegação.
- Evite sinais "em construção" — um link desnecessário certamente desaparecerá.
- Os usuários preferem não "rolar" a tela. Informações importantes deveriam ser colocadas em dimensões de uma típica janela de navegação.
- Menus de navegação e barras de título devem ser desenhados de forma consistente e estar disponíveis em todas as páginas que são acessadas pelo usuário. O projeto não deve depender de funções do navegador para ajudar na navegação.
- A estética jamais deve suplantiar a funcionalidade. Por exemplo, um botão simples poderia ser uma opção de navegação melhor que um botão esteticamente charmoso, mas que é apenas uma vaga imagem ou ícone cujo objetivo não é claro.
- As opções de navegação devem ser óbvias, mesmo para o usuário casual. O usuário não deve ter de ficar procurando na tela para determinar como fazer o link com outros conteúdos ou serviços.

Uma interface bem projetada aumenta a percepção do usuário do conteúdo ou serviços fornecidos pelo site. Ela não precisa, necessariamente, ser chamativa, mas sempre deve ser bem estruturada e ergonomicamente sólida.

11.5.2 Fluxo de trabalho de projeto de interfaces para WebApps

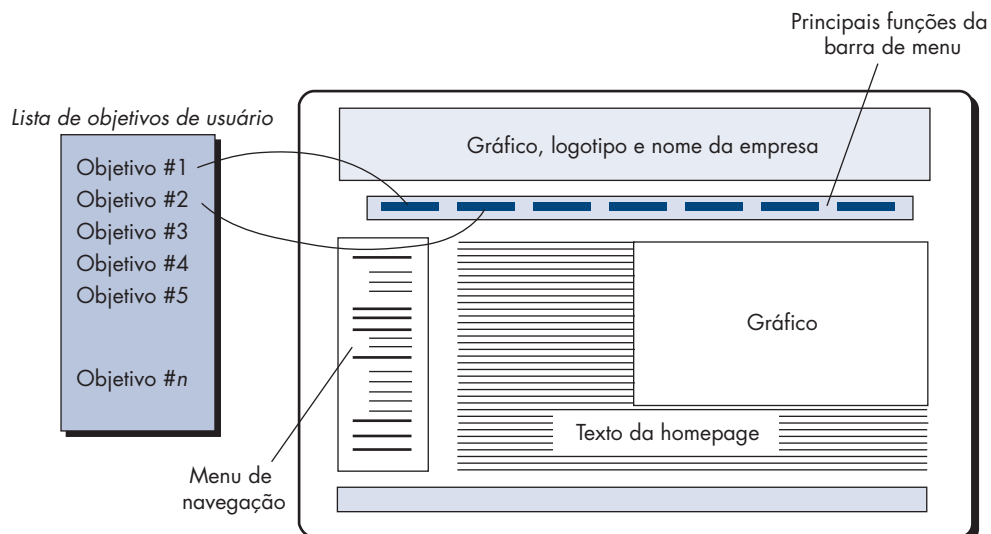
No início do capítulo citei que o projeto de interfaces do usuário começa com a identificação dos requisitos do usuário, de tarefas e dos ambientes. Uma vez que as tarefas de usuário tenham sido identificadas, cenários de usuário (casos de uso) são criados e analisados para definir um conjunto de ações e objetos de interface.

Informações contidas no modelo de requisitos formam a base para a criação de um layout de tela que represente o design gráfico e o posicionamento de ícones, a definição de texto de tela descritivo, a especificação e a colocação de títulos para as janelas, bem como a especificação de itens de menu principais e secundários. São usadas então ferramentas para prototipagem e, por fim, a implementação do modelo de projeto para interface. As tarefas a seguir representam um fluxo de trabalho rudimentar no projeto de interfaces para WebApps:

- 1. Revise as informações contidas no modelo de requisitos e refine conforme necessário.**
- 2. Desenvolva um esboço do layout para interfaces WebApp.** Um protótipo de interface (incluindo seu layout) poderia ter sido desenvolvido como parte da atividade de modelagem de requisitos. Se o layout já existir, deve ser revisto e refinado conforme necessário. Se o layout da interface não tiver sido desenvolvido, devemos trabalhar com os interessados para desenvolvê-lo nesta oportunidade. Um esboço de layout preliminar é mostrado na Figura 11.4.
- 3. Mapeie os objetivos do usuário em ações de interface específicas.** Para a grande maioria das WebApps, o usuário terá um conjunto relativamente pequeno de objetivos primários. Esses devem ser mapeados em ações de interface específicas conforme mostra a Figura 11.4. Em essência, temos de responder a seguinte pergunta: “Como a interface habilita o usuário a cumprir cada objetivo?”.
- 4. Defina um conjunto de tarefas de usuário que está associado a cada ação.** Cada ação de interface (por exemplo, “comprar um produto”) está associada a um conjunto de tarefas de usuário. Tais tarefas foram identificadas durante a modelagem de requisitos. Durante o projeto, devem ser mapeadas em interações específicas que englobam questões de navegação, objetos de conteúdo e funções WebApp.

FIGURA 11.4

Mapeamento dos objetivos de usuário em ações de interface



5. **Imagens de tela storyboard para cada ação de interface.** À medida que cada ação é considerada, uma sequência de imagens de storyboard (imagens de tela) deve ser criada para representar como a interface responde à interação com o usuário. Devem-se identificar objetos de conteúdo (mesmo que ainda não tenham sido projetados e desenvolvidos), a funcionalidade da WebApp deve ser mostrada e links de navegação devem ser indicados.
6. **Refine o layout de interface e storyboards usando entrada do projeto estético.** Na maioria dos casos, você será responsável por layout e storyboarding preliminar, mas o aspecto estético para um importante site comercial em geral é desenvolvido por profissionais artísticos e não por técnicos. O projeto estético (Capítulo 13) é integrado com o trabalho realizado pelo projetista de interface.
7. **Identifique objetos de interface com o usuário necessários para implementar a interface.** Essa tarefa poderia exigir uma busca em uma biblioteca de objetos para encontrar aqueles objetos (classes) reutilizáveis apropriados para a interface para WebApp. Além disso, quaisquer classes personalizadas são especificadas neste momento.
8. **Desenvolva uma representação procedural da interação do usuário com a interface.** Essa tarefa opcional usa diagramas de sequências UML e/ou diagramas de atividades (Apêndice 1) para representar o fluxo de atividades (e decisões) que ocorrem à medida que o usuário interage com a WebApp.
9. **Desenvolva uma representação comportamental da interface.** Essa tarefa opcional faz uso de diagramas de estados UML (Apêndice 1) para representar transições de estados e os eventos que os provocam. São definidos mecanismos de controle (isto é, os objetos e as ações disponíveis para o usuário alterar um estado da WebApp).
10. **Descreva o layout da interface para cada estado.** Usando as informações de projeto desenvolvidas nas Tarefas 2 e 5, associe a layout ou imagens de tela específicos a cada estado da WebApp descrito na Tarefa 8.
11. **Refine e revise o modelo de projeto de interface.** A revisão da interface deve se concentrar na usabilidade.

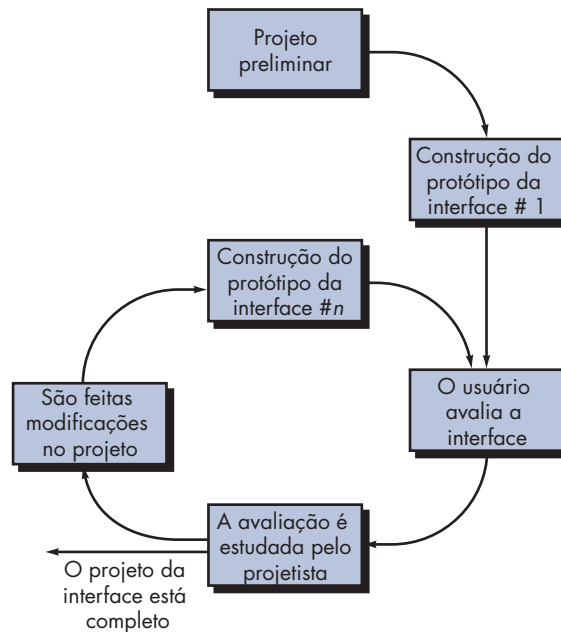
É importante notar que o conjunto final de tarefas escolhido deve ser adaptado às exigências especiais da aplicação a ser construída.

11.6 AVALIAÇÃO DE PROJETO

Assim que tiver criado um protótipo operacional de interface do usuário, ele deve ser avaliado para determinar se atende os requisitos do usuário. A avaliação pode abranger um espectro de formalidade que vai de um *test-drive* informal em que um usuário fornece feedback imediato ao estudo formalmente projetado que usa os métodos estatísticos para a avaliação de questionários preenchidos por uma população de usuários finais.

O ciclo de avaliação de interfaces do usuário assume a forma indicada na Figura 11.5. Após a finalização do modelo de projeto, cria-se um protótipo de primeiro nível. O protótipo é avaliado pelo usuário¹¹, que nos fornece comentários diretos sobre a eficácia da interface. Além disso, se forem usadas técnicas de avaliação formais (por exemplo, questionários, formulários de avaliação), podemos extrair informações desses dados (por exemplo, 80% de todos os usuários não gostam do mecanismo para salvar arquivos de dados). São feitas modificações de projeto baseadas nas informações fornecidas pelos usuários e é criado o protótipo do nível seguinte. O ciclo de avaliação continua até que nenhuma modificação adicional seja necessária para o projeto de interfaces.

11 É importante notar que os especialistas em ergonomia e projeto de interfaces também podem realizar revisões da interface. Essas revisões são denominadas *avaliações heurísticas* ou *revisões cognitivas*.

FIGURA 11.5**O ciclo de avaliação
de projeto de interfaces**

A abordagem de prototipagem é efetiva, mas é possível avaliar a qualidade de uma interface do usuário antes de um protótipo ser construído? Se forem identificados e corrigidos possíveis problemas precocemente, o número de laços realizados no ciclo de avaliação será reduzido, e o tempo de desenvolvimento será abreviado. Se um modelo de projeto da interface tiver sido criado, uma série de critérios de avaliação [Mor81] poderá ser aplicada durante revisões de projeto precoces:

1. A duração e a complexidade do modelo de requisitos ou a especificação por escrito do sistema e sua interface dão uma indicação do volume de aprendizado exigido dos usuários do sistema.
2. O número de tarefas de usuário especificado e o número médio de ações por tarefa indicam o tempo de interação e a eficiência geral do sistema.
3. O número de ações, tarefas e estados do sistema indicados pelo modelo de projeto sugerem a carga de memória necessária por parte dos usuários do sistema.
4. O estilo da interface, recursos de ajuda e o protocolo de tratamento de erros dão uma indicação geral da complexidade da interface e do grau de aceitação por parte dos usuários.

Assim que o primeiro protótipo tiver sido construído, podemos coletar uma variedade de dados qualitativos e quantitativos que irão auxiliar na avaliação da interface. Para a coleta de dados qualitativos, podemos distribuir questionários aos usuários do protótipo. Entre as perguntas a ser feitas, temos: (1) respostas simples SIM/NÃO, (2) respostas numéricas, (3) resposta em padrão de escala (subjativa), (4) escalas de Likert (como, por exemplo, concordo plenamente, concordo parcialmente), (5) resposta porcentual (subjativa) ou (6) aberta.

Se forem desejados dados quantitativos, pode ser realizada uma espécie de análise de estudo de tempos. Os usuários são observados durante a interação, e dados — como uma série de tarefas corretamente completadas durante um período de tempo padrão, frequência de ações, sequência de ações, tempo gasto “observando” a tela, número e tipos de erros, tempo de recuperação de erros, tempo gasto usando o sistema de ajuda e o número de referências de ajuda por período de tempo padrão — são coletados e usados como um guia para modificação da interface.

Uma discussão completa dos métodos de avaliação de interfaces do usuário está fora do escopo deste livro. Para maiores informações, veja [Hac98] e [Sto05].

11.7 RESUMO

A interface do usuário é discutivelmente o elemento mais importante de um produto ou sistema computacional. Se a interface for mal projetada, a capacidade de o usuário aproveitar todo o poder computacional e conteúdo de informações de uma aplicação pode ser seriamente afetada. Na realidade, uma interface fraca pode fazer com que uma aplicação, em outros aspectos bem projetada e solidamente implementada, falhe.

Três importantes princípios orientam o projeto de interfaces do usuário eficazes: (1) deixar o usuário no comando, (2) reduzir a carga de memória do usuário e (3) tornar a interface consistente. Para alcançar uma interface que observe esses princípios, deve ser realizado um processo de projeto organizado.

O desenvolvimento de uma interface do usuário começa com uma série de tarefas de análise. A análise dos usuários define os perfis de vários usuários finais e é reunida com base em uma série de fontes técnicas e comerciais. A análise de tarefas define as tarefas e ações de usuários usando uma abordagem de refinamento ou orientada a objetos, aplicando casos de uso, elaboração de tarefa e objetos, análise de fluxos de trabalho e representações hierárquicas de tarefas para entender completamente a interação homem-computador. A análise do ambiente identifica as estruturas físicas e sociais em que a interface deve operar.

Uma vez identificadas as tarefas, são criados e analisados cenários de usuário para definir um conjunto de ações e objetos de interface. Isso fornece uma base para a criação de um layout de tela que represente o design gráfico e o posicionamento de ícones, a definição de texto de tela descritivo, a especificação e a colocação de títulos para as janelas, bem como a especificação de itens de menu principais e secundários. Questões de projeto como tempo de resposta, estrutura de comandos e ações, tratamento de erros e recursos de ajuda são consideradas à medida que o modelo de projeto é refinado. Uma grande variedade de ferramentas de implementação é usada para construir um protótipo para avaliação por parte do usuário.

Assim como ocorre no projeto de interfaces para software convencional, o projeto de interfaces para WebApps descreve a estrutura e a organização de uma interface do usuário e abrange a representação do layout de tela, a definição dos modos de interação e a descrição de mecanismos de navegação. Um conjunto de princípios para o projeto de interfaces e um fluxo de trabalho para projeto de interfaces orientam o projetista de WebApps quando o layout e mecanismos de controle da interface são desenhados.

A interface do usuário é a janela para o software. Em muitos casos, ela molda a percepção do usuário quanto à qualidade de um sistema. Se a “janela” for embaçada, ondulada ou quebrada, o usuário poderá rejeitar um sistema computacional que de outra forma seria considerado poderoso.

PROBLEMAS E PONTOS A PONDERAR

11.1. Descreva a pior interface com a qual você já tenha trabalhado até hoje e critique-a em relação aos conceitos introduzidos neste capítulo. Descreva a melhor interface com a qual já tenha trabalhado até hoje e critique-a em relação aos conceitos introduzidos neste capítulo.

11.2. Desenvolva mais dois princípios de projeto que “deixem o usuário no comando”.

11.3. Desenvolva mais dois princípios de projeto que “reduzam a carga de memória do usuário”.

11.4. Desenvolva mais dois princípios de projeto que “tornem a interface consistente”.

11.5. Considere uma das seguintes aplicações interativas (ou uma aplicação designada pelo seu professor):

- a. Um sistema de editoração eletrônica
- b. Um sistema de projeto auxiliado por computador
- c. Um sistema de projeto de interiores (conforme descrito na Seção 11.3.2)
- d. Um sistema automatizado de matrícula de uma universidade
- e. Um sistema de gerenciamento de bibliotecas
- f. Uma urna eletrônica baseada na Internet para eleições públicas
- g. Um sistema de *home banking*
- h. Uma aplicação interativa designada pelo seu professor

Desenvolva um modelo de usuário, modelo de projeto, modelo mental e um modelo de implementação para qualquer um desses sistemas.

11.6. Realize uma análise detalhada das tarefas para qualquer um dos sistemas enumerados no Problema 11.5. Use uma abordagem de refinamento ou orientada a objetos.

11.7. Acrescente pelo menos cinco outras questões à lista desenvolvida para análise de conteúdo da Seção 11.3.3.

11.8. Continuando o Problema 11.5, defina objetos e ações de interface para a aplicação que você escolheu. Identifique cada tipo de objeto.

11.9. Desenvolva um conjunto de layouts de tela com uma definição de itens de menu primários e secundários para o sistema escolhido no Problema 11.5.

11.10. Desenvolva um conjunto de layouts de tela com uma definição de itens de menu primários e secundários para o sistema *CasaSegura*. Você pode optar por uma abordagem diferente daquela indicada para o layout de tela da Figura 11.3.

11.11. Descreva sua abordagem para recursos de ajuda ao usuário para o modelo de projeto de análise de tarefas e para a análise de tarefas realizada como parte dos Problemas 11.5 a 11.8.

11.12. Dê alguns exemplos que ilustrem por que a variabilidade no tempo de resposta pode ser um problema.

11.13. Desenvolva uma abordagem que integraria automaticamente mensagens de erro e um recurso de ajuda ao usuário. Isto é, o sistema reconheceria automaticamente o tipo de erro e forneceria uma janela de ajuda com sugestões para corrigi-lo. Realize um projeto de software razoavelmente completo que considere estruturas de dados e algoritmos apropriados.

11.14. Desenvolva um questionário para avaliação de interfaces contendo 20 perguntas genéricas que se aplicariam à maioria das interfaces. Faça com que 10 companheiros de classe completem o questionário para um sistema interativo que todos usarão. Sintetize os resultados e relate-os à sua classe.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Embora este livro não seja especificamente sobre interfaces homem-computador, muito daquilo que Donald Norman (*The Design of Everyday Things*, reedição, Currency/Doubleday, 1990) tem a dizer sobre a psicologia de projeto efetivo se aplica a uma interface do usuário. É uma leitura recomendada a todos que estejam empenhados em desenvolver um projeto de interface de alta qualidade.

As interfaces gráficas do usuário são onipresentes no mundo moderno da computação. Seja ela um caixa eletrônico, um telefone celular, um painel eletrônico de comandos em um automóvel, um site ou uma aplicação comercial, a interface do usuário oferece uma janela para o software. É por essa razão que há inúmeros livros que tratam de projeto de interfaces. Butow (*User Interface Design for Mere Mortals*, Addison-Wesley, 2007), Galitz (*The Essential Guide to User Interface Design*, 3. ed., Wiley, 2007), Lehtonen e seus colegas (*Personal Content Experience: Managing Digital Life in the Mobile Age*, Wiley-Interscience, 2007), Cooper e seus

colegas (*About Face 3: The Essentials of Interaction Design*, 3. ed., Wiley, 2007), Ballard (*Designing the Mobile User Experience*, Wiley, 2007), Nielsen (*Coordinating User Interfaces for Consistency*, Morgan-Kaufmann, 2006), Lauesen (*User Interface Design: A Software Engineering Perspective*, Addison-Wesley, 2005), Barfield (*The User Interface: Concepts and Design*, Bosko Books, 2004) todos discutem temas como usabilidade, conceitos, princípios e técnicas de projeto para interfaces do usuário e contêm muitos exemplos úteis.

Livros mais antigos como os de Beyer e Holtzblatt (*Contextual Design: A Customer Centered Approach to Systems Design*, Morgan-Kaufmann, 2002), Raskin (*The Humane Interface*, Addison-Wesley, 2000), Constantine e Lockwood (*Software for Use*, ACM Press, 1999) e Mayhew (*The Usability Engineering Lifecycle*, Morgan-Kaufmann, 1999) apresentam tratados que fornecem diretrizes e princípios de projeto adicionais, bem como sugestões para levantamento de necessidades, modelagem de projeto, implementação e testes de interfaces. Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kaufmann, 2000) dá uma útil orientação para aqueles que aprendem mais efetivamente por meio do exame de contraexemplos. Um agradável livro de Cooper (*The Inmates Are Running the Asylum*, Sams Publishing, 1999) discute por que os produtos de alta tecnologia nos deixam loucos e como projetar algum deles que não tenham esse efeito.

Uma ampla gama de fontes de informação sobre projeto de interfaces se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevante para o projeto de interfaces, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

12

PROJETO BASEADO EM PADRÕES

CONCEITOS-CHAVE

erros de projeto .	326
estruturas	320
forças.	318
granularidade . . .	334
linguagem de padrões	321
padrões	
comportamentais. 327	
criacionais	319
de arquitetura. .	327
de componentes. 329	
estruturais	319
generativos . . .	318
para interfaces do usuário	333
para WebApps .	333

Todos já nos deparamos com um problema de projeto e, silenciosamente, pensamos: Será que alguém já desenvolveu uma solução para este problema? A resposta é quase sempre — sim! O problema é encontrar a solução; garantir que, de fato, se adapte ao problema em questão; entender as restrições que talvez limitem a maneira pela qual a solução é aplicada e, por fim, traduzir a solução proposta para seu ambiente de projeto.

Mas o que acontece se a solução fosse de alguma forma codificada? E se existisse uma maneira padronizada de descrever um problema (de tal forma que pudéssemos pesquisá-la) e um método organizado para representar a solução para o problema? Acaba evidenciando-se que problemas de software foram codificados e descritos usando-se um template padronizado e que foram propostas soluções (juntamente com restrições) para eles. Denominado *padrões de projeto*, esse método codificado para descrição de problemas e suas soluções permite aos profissionais da engenharia de software adquirir conhecimento de projeto para possibilitar que ele seja reutilizado.

O início da história dos padrões de software não se inicia com um cientista da computação, mas com um arquiteto, Christopher Alexander, que reconheceu o fato de um conjunto de problemas recorrentes ser encontrado toda vez que um edifício era projetado. Ele caracterizou esses problemas recorrentes e suas soluções como *padrões*, descrevendo-os da seguinte maneira [Ale77]:

PANORAMA

O que é? O projeto baseado em padrões cria uma nova aplicação através da busca de um conjunto de soluções comprovadas para um conjunto de problemas claramente delineados. Cada problema (e sua solução) é descrito por um padrão de projeto que foi catalogado e investigado por outros engenheiros de software que depararam com o problema e implementaram a solução ao projetarem outras aplicações. Cada padrão de projeto nos oferece uma abordagem comprovada para parte do problema a ser resolvido.

Quem realiza? Um engenheiro de software examina cada problema que surge para uma nova aplicação e tenta encontrar uma solução relevante por meio de pesquisa em um ou mais repositórios de padrões.

Por que é importante? Você já ouviu a frase “reinventar a roda”? Ela ocorre toda hora em desenvolvimento de software e é uma perda de tempo e energia. Ao usarmos padrões de projeto, podemos encontrar uma solução comprovada para um problema específico. À medida que cada padrão é aplicado, são integradas soluções, e a aplicação a ser construída se aproxima cada vez mais de um projeto completo.

Quais são as etapas envolvidas? O modelo de requisitos é examinado para isolar o conjun-

to hierárquico de problemas a ser resolvido. O espaço de problemas é subdividido de modo que subconjuntos de problemas associados a funções e características de software específicas possam ser identificados. Os problemas também podem ser organizados por tipo: de arquitetura, de componentes, algorítmicos, de interfaces do usuário etc. Uma vez definido um subconjunto de problemas, pesquisam-se um ou mais repositórios de padrões para determinar se um padrão de projeto existente, representado em um nível de abstração apropriado, existe. Padrões aplicáveis são adaptados às necessidades específicas do software a ser construído. A solução de problemas personalizados é aplicada em situações em que não foi encontrado nenhum padrão.

Qual é o artefato? É desenvolvido um modelo de projeto que represente a estrutura da arquitetura, a interface do usuário e detalhes em nível de componentes.

Como garantir que o trabalho foi realizado corretamente? À medida que cada padrão de projeto é traduzido em algum elemento do modelo de projeto, os artefatos são revistos em termos de clareza, correção, completude e consistência em relação às necessidades e entre si.

Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente e então descreve o cerne de uma solução para aquele problema para podermos usar a solução repetidamente um milhão de vezes sem jamais ter de fazer a mesma coisa duas vezes.

As ideias de Alexander foram primeiro traduzidas para o mundo do software em livros como os de Gamma [Gam95], Buschmann [Bus96] e seus vários colegas¹. Hoje, existem dezenas de repositórios de padrões, e projetos baseados em padrões podem ser aplicados em diversos domínios de aplicação.

12.1 PADRÃO DE PROJETO

PONTO-CHAVE

Forças são aquelas características do problema e atributos de uma solução que restringem a maneira pela qual o projeto pode ser desenvolvido.

Um padrão de projeto pode ser caracterizado como “uma regra de três partes que expressa uma relação entre um contexto, um problema e uma solução” [Ale79]. Para projeto de software, o *contexto* permite ao leitor compreender o ambiente em que o problema reside e qual solução poderia ser apropriada nesse ambiente. Um conjunto de requisitos, incluindo limitações e restrições, atua como um *sistema de forças* que influencia a maneira pela qual o problema pode ser interpretado em seu contexto e como a solução pode ser efetivamente aplicada.

Para compreendermos melhor esses conceitos, consideremos a situação² em que uma pessoa tenha de viajar de Nova York a Los Angeles. Nesse contexto, a viagem ocorrerá em um país industrializado (os Estados Unidos), usando uma infraestrutura de transporte existente (por exemplo, estradas, linhas aéreas, ferrovias). O sistema de forças que afetará a maneira pela qual o problema de viagem é solucionado abrangerá: com que rapidez a pessoa quer ir de Nova York a LA, se a viagem incluirá ou não pontos turísticos ou de parada, quanto dinheiro a pessoa pode gastar, se a viagem se destina a cumprir um propósito específico e os veículos pessoais que a pessoa tem à sua disposição. Dadas essas forças, o problema (viajar de Nova York a LA) pode ser mais bem definido. Por exemplo, uma investigação (levantamento de requisitos) indica que a pessoa tem pouquíssimo dinheiro, tem apenas uma bicicleta (e é um ciclista entusiasta), quer fazer a viagem para arrecadar fundos para sua instituição de caridade preferida e tem muito tempo disponível. A solução para o problema, dado o contexto e o sistema de forças, poderia ser uma viagem de bicicleta atravessando o país de ponta a ponta. Se as forças fossem diferentes (por exemplo, o tempo de viagem deve ser minimizado e o propósito da viagem é uma reunião de negócios), talvez fosse mais apropriada uma outra solução.

É razoável argumentar que a maioria dos problemas possui várias soluções, porém uma solução é efetiva apenas se for apropriada no contexto do problema existente.

É o sistema de forças que faz com que um projetista escolha uma solução específica. O intuito é fornecer uma solução que melhor atenda o sistema de forças, mesmo quando essas forças forem contrárias. Por fim, toda solução tem consequências que poderiam ter um impacto sobre outros aspectos do software e ela própria poderia fazer parte do sistema de forças para outros problemas a ser resolvidos no sistema mais amplo. Coplien [Cop05] caracteriza um padrão de projeto eficaz da seguinte maneira:

- *Ele soluciona um problema*: Os padrões capturam soluções, não apenas estratégias ou princípios abstratos.
- *Ele é um conceito comprovado*: Os padrões apreendem soluções com um histórico, não teorias ou especulação.
- *Uma solução não é óbvia*: Muitas técnicas para resolução de problemas (como paradigmas ou métodos de projeto de software) tentam obter soluções com base nos primeiros princípios. Os melhores padrões *geram* uma solução para um problema indiretamente — uma abordagem necessária para os problemas mais difíceis de projeto.

“Nossa responsabilidade é fazer o que podemos, aprender o que podemos, aperfeiçoar as soluções e passá-las adiante.”

Richard P. Feynman

¹ Existem discussões iniciais de padrões de software, porém esses dois clássicos foram os primeiros tratados coesos sobre o assunto.

² Esse exemplo foi adaptado de [Cor98].

- *Ele descreve uma relação:* Os padrões não apenas descrevem módulos, como também estruturas e mecanismos de sistema mais profundos.
- *O padrão possui um componente humano significativo (minimizar a intervenção humana).* Todo software visa atender o conforto humano ou a qualidade de vida; os melhores padrões apelam explicitamente à estética e utilidade.

Declarado de uma forma ainda mais pragmática, um padrão de projeto adequado captura conhecimento de projeto pragmático e ganha com muito suor, de forma que outros possam reutilizar esse conhecimento “uma milhão de vezes sem jamais ter de fazer a mesma coisa duas vezes”. Um padrão de projeto evita que tenhamos de “reinventar a roda”, ou pior ainda, inventar uma “nova roda” que não será perfeitamente redonda, será muito pequena para o uso pretendido e muito estreita para o terreno onde irá rodar. Os padrões de projeto, se usados de maneira efetiva, invariavelmente o tornarão um melhor projetista de software.

12.1.1 Tipos de padrões

Uma das razões para os engenheiros de software interessarem-se (e intrigados) em padrões de projeto é o fato de os seres humanos serem inerentemente bons no reconhecimento de padrões. Se não fossemos, teríamos parado no tempo e no espaço — incapazes de aprender de experiência passada, desinteressados em nos aventurarmos devido à nossa inabilidade de reconhecer situações que talvez nos levassem a correr altos riscos, transtornados por um mundo que parece não ter regularidade ou consistência lógica. Felizmente, nada disso acontece porque efetivamente reconhecemos padrões em praticamente todos os aspectos de nossas vidas.

No mundo real, os padrões que reconhecemos são aprendidos ao longo de toda uma vida de experiência. Reconhecemos instantaneamente e compreendemos inerentemente seus significados e como eles poderiam ser usados. Alguns desses padrões nos dão uma melhor visão do fenômeno da recorrência. Por exemplo, você está voltando do trabalho para casa na Marginal quando seu sistema de navegação (ou o rádio do carro) informa-lhe que um grave acidente ocorreu na Marginal no sentido oposto. Você se encontra a 6 quilômetros do acidente, porém já começa a perceber tráfego lento, reconhecendo um padrão que chamaremos **RubberNecking (olhar com curiosidade)**. Os motoristas deslocando-se na pista expressa na sua direção estão diminuindo de velocidade, para ter uma melhor visão do que aconteceu no sentido contrário. O padrão **RubberNecking** produz resultados notavelmente previsíveis (um congestionamento), mas nada mais faz que descrever um fenômeno. No jargão dos padrões, ele poderia ser denominado padrão *não generativo* pois descreve um contexto e um problema, mas não fornece nenhuma solução explícita.

Quando são considerados padrões de projeto de software, nos esforçamos para identificar e documentar padrões *generativos*. Ou seja, identificamos um padrão que descreve um aspecto importante e repetível de um sistema e que nos dá uma maneira de construir esse aspecto em um sistema de forças que são únicas em determinado contexto. Em um ambiente ideal, um conjunto de padrões de projeto generativos poderia ser usado para “gerar” uma aplicação ou sistema computacional cuja arquitetura permitisse que se adaptasse à mudança. Algumas vezes chamada *generatividade*, “a aplicação sucessiva de vários padrões, cada um deles encapsulando seu próprio problema e forças, se desdobra em uma solução mais ampla que emerge indiretamente como resultado das soluções menores” [App00].

Os padrões de projeto abrangem um amplo espectro de abstração e aplicação. Os *padrões de arquitetura* descrevem problemas de projeto de caráter amplo e diverso resolvidos usando-se uma abordagem estrutural. Os *padrões de dados* descrevem problemas orientados a dados recorrentes e as soluções de modelagem de dados que podem ser usadas para resolvê-los. Os *padrões de componentes* (também conhecidos como *padrões de projeto*) tratam de problemas associados ao desenvolvimento de subsistemas e componentes, a maneira através da qual eles se comunicam entre si e seu posicionamento em uma arquitetura maior. Os *padrões de projeto de interfaces* descrevem problemas comuns de interface do usuário e suas soluções com

PONTO-CHAVE

Um padrão “generativo” descreve o problema, um contexto e forças, mas também descreve uma solução pragmática para o problema.

um sistema de forças que incluía características específicas de usuários finais. Os *padrões para WebApp* tratam de um conjunto de problemas encontrados ao se construir WebApps e em geral incorpora muitas das demais categorias de padrões que acabamos de mencionar. Em um nível de abstração mais baixo, os *idiomas* descrevem como implementar todos ou parte de um algoritmo específico ou estrutura de dados para um componente de software no contexto de uma linguagem de programação específica.

Em seu livro seminal sobre padrões de projeto, Gamma e seus colegas³ [Gam95] focalizam três tipos de padrões particularmente relevantes em projetos orientados a objetos: padrões criacionais, padrões estruturais e padrões comportamentais.

 Existe uma maneira de classificar tipos de padrão?



Padrões criacionais, estruturais e comportamentais

Foi proposta uma ampla variedade de padrões de projeto que se encaixam nas categorias criacional, estrutural e comportamental e podem ser encontradas na Web.

A Wikipedia (www.wikipedia.org) cita a seguinte relação:

Padrões criacionais

- **Padrão de fábrica abstrata (factory pattern):** centraliza a decisão de que fábrica instanciar.
- **Padrão de métodos de fábrica (factory method pattern):** centraliza a criação de um objeto de um tipo específico escolhendo uma de várias implementações.
- **Padrão construtor (builder pattern):** separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.
- **Padrão de protótipo (prototype pattern):** usado quando o custo inerente de criação de um novo objeto da maneira padrão (por exemplo, usando a palavra-chave “new”) é proibitivo para uma dada aplicação.
- **Padrão único (singleton pattern):** restringe a instancição de uma classe a um único objeto.

Padrões estruturais

- **Padrão adaptador (adapter pattern):** “adapta” uma interface de uma classe para uma que um cliente espera.
- **Padrão de agregação (aggregate pattern):** uma versão do padrão de composição com métodos para agregação de objetos filhos.
- **Padrão ponte (bridge pattern):** desacopla uma abstração de sua implementação de modo que as duas possam variar independentemente.
- **Padrão de composição (composite pattern):** uma estrutura de objetos em forma de árvore onde cada objeto possui a mesma interface.
- **Padrão container (container pattern):** cria objetos com o propósito exclusivo de conter outros objetos e gerenciá-los.

- **Padrão proxy (proxy pattern):** uma classe atuando como uma interface para outra.
- **tubos e filtros (pipes and filters patterns):** uma cadeia de processos onde a saída de cada processo é a entrada do seguinte.

Padrões comportamentais

- **Padrão de cadeia de responsabilidades (chain of responsibility pattern):** Objetos e comando são manipulados ou passados para outros objetos através de objetos de processamento com lógica.
- **Padrão de comandos (command pattern):** Objetos de comando encapsulam uma ação e seus parâmetros.
- **Escutador de eventos (event listener):** Os dados são distribuídos a objetos que se registraram para recebê-los.
- **Padrão interpretador (interpreter pattern):** Implementa uma linguagem computacional especializada para resolver rapidamente um conjunto de problemas específicos.
- **Padrão iterador (iterator pattern):** Os iteradores são usados para acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.
- **Padrão mediador (mediator pattern):** fornece uma interface unificada para um conjunto de interfaces em um subsistema.
- **Padrão visitante (visitor pattern):** Uma forma de separar um algoritmo de um objeto.
- **Padrão visitante para atendimento único (single-serving visitor pattern):** Otimiza a implementação de um visitante que é alocado, usado apenas uma vez e depois deletado.
- **Padrão visitante hierárquico (hierarchical visitor pattern):** Fornece uma forma de visitar todos os nós em uma estrutura de dados hierárquica como, por exemplo, uma árvore.

Descrições detalhadas de cada um desses padrões podem ser obtidas via links em www.wikipedia.org.

3 Gamma e seus colegas [Gam95] são normalmente conhecidos como a “Gang of Four” (GoF) na literatura sobre padrões.

Os *padrões criacionais* concentram-se na “criação, composição e representação” de objetos. Gamma e seus colegas [Gam95] observam que os padrões criacionais “encapsulam o conhecimento sobre quais classes concretas o sistema usa”, porém, ao mesmo tempo, “ocultam como instâncias dessas classes são criadas e combinadas”. Os padrões criacionais dispõem de mecanismos que facilitam a instanciação de objetos em um sistema e impõem “restrições sobre o tipo e número de objetos que podem ser criados em um sistema” [Maa07].

Os *padrões estruturais* focalizam problemas e soluções associadas a como classes e objetos são organizados e integrados para construir uma estrutura maior. Em essência, ajudam a estabelecer relações entre entidades em um sistema. Por exemplo, padrões estruturais que se concentram em questões orientadas a classes poderiam fornecer mecanismos de herança que levem a interfaces de programa mais eficazes. Padrões estruturais que se concentram em objetos sugerem técnicas para combinar objetos em outros objetos ou integrar objetos em uma estrutura maior.

Os *padrões comportamentais* tratam de problemas associados à atribuição de responsabilidade entre objetos e a maneira pela qual a comunicação é efetuada entre objetos.

12.1.2 Estruturas de uso de padrões de projeto

Os próprios padrões talvez não sejam suficientes para desenvolver um projeto completo. Em alguns casos pode ser necessário fornecer uma infraestrutura mínima específica a uma implementação, para trabalho de projeto. Podemos selecionar uma “*mini-arquitetura reutilizável* que fornece a estrutura genérica e o comportamento para uma família de abstrações de software, juntamente com um contexto... Que especifica sua colaboração e uso em um domínio de dados” [Amb98].

Essa estrutura de uso de padrões⁴ não é um padrão de arquitetura, mas sim um esqueleto com um conjunto de “pontos de conexão” (também chamados *ganchos* e *encaixes*) que permitem-lhe ser adaptada a um domínio de problemas específico. Os pontos de conexão possibilitam que integremos ao esqueleto classes ou funcionalidades específicas de um problema. Em um contexto orientado a objetos, uma estrutura é um conjunto de classes que cooperam entre si.

Gamma e seus colegas [Gam95] descrevem as diferenças entre padrões de projeto e estruturas de uso de padrões da seguinte maneira:

1. *Os padrões de projeto são mais abstratos que as estruturas de uso.* As estruturas de uso de padrões podem ser incorporadas ao código, mas apenas *exemplos* de padrões podem ser incorporados ao código. Um ponto forte das estruturas de uso de padrões é que podem ser escritas em linguagens de programação e não só estudadas, mas executadas e reutilizadas diretamente.
2. *Os padrões de projeto são elementos arquiteturais menores que as estruturas de uso de padrões.* Essas estruturas podem conter vários padrões de projeto, mas o contrário jamais é verdadeiro.
3. *Os padrões de projeto são menos especializados que as estruturas de uso de padrões.* Elas apresentam um domínio de aplicação particular. Diferentemente, os padrões de projeto podem ser usados em praticamente qualquer tipo de aplicação. Embora sejam possíveis padrões de projeto mais especializados, mesmos estes não ditariam uma arquitetura de aplicação.

Em essência, o projetista de uma estrutura de uso de padrão argumentará que uma *mini-arquitetura reutilizável* se aplica a todo software a ser desenvolvido em um domínio de aplicação limitado. Para serem mais efetivas, essas estruturas são aplicadas sem nenhuma alteração. Podem-se acrescentar outros elementos de projeto, mas apenas através de pontos de conexão que permitem ao projetista dar corpo ao esqueleto dessas estruturas.

12.1.3 Descrição de padrões

O projeto baseado em padrões começa com o reconhecimento de padrões na aplicação que se pretende construir, continua com a pesquisa para determinar se outros trataram o padrão

PONTO-CHAVE

Estrutura de uso de padrões é uma “mini-arquitetura” reutilizável que serve como base e a partir da qual outros padrões de projeto podem ser aplicados.

⁴ N. de R.T.: Uma estrutura de uso de padrões inclui aspectos estáticos/estruturais e dinâmicos/comportamentais que dão suporte ao uso dos padrões de projeto.

e termina com a aplicação de um padrão apropriado para o problema em questão. Em geral, a segunda dessas três tarefas é a mais difícil. Como encontrar padrões que atendam nossas necessidades?

A resposta deve depender da comunicação efetiva do problema que o padrão trata, do contexto em que o padrão reside, do sistema de forças que molda o contexto e da solução proposta. Para transmitir essas informações de forma inequívoca, é necessário um formulário ou template padronizado para descrições de padrão. Embora tenham sido propostos vários templates de padrão distintos, quase todos contêm um subconjunto principal do conteúdo sugerido por Gamma e seus colegas [Gam95]. No quadro a seguir é mostrado um template de padrão simplificado.

INFORMAÇÕES

**Template de padrões de projeto**

Nome do padrão — descreve a essência do padrão em um nome curto mas expressivo.

Problema — descreve o problema de que o padrão trata.

Motivação — dá um exemplo do problema.

Contexto — descreve o ambiente onde o problema reside incluindo o domínio de aplicação.

Forças — enumera o sistema de forças que afetam a maneira pela qual o problema deve ser resolvido; inclui uma discussão das limitações e restrições que devem ser consideradas.

Solução — fornece uma descrição detalhada de uma solução proposta para o problema.

Objetivo — descreve o padrão e o que ele faz.

Colaborações — descrevem como outros padrões contribuem para uma solução.

Consequências — descrevem os possíveis prós e contras que devem ser considerados quando o padrão é implementado e as consequências do uso do padrão.

Implementação — identifica questões especiais que devem ser consideradas ao se implementar o padrão.

Usos conhecidos — dá exemplos de usos práticos do padrão de projeto em aplicações reais.

Padrões relacionados — remetem a padrões de projeto relacionados.

“Os padrões são semiprontos — isso significa que sempre temos de finalizá-los e adaptá-los ao nosso ambiente.”

Martin Fowler

Os nomes de padrões de projeto devem ser escolhidos com cuidado. Um dos principais problemas técnicos em projeto baseado em padrões é a inability de encontrar padrões existentes quando há centenas ou milhares de candidatos. A busca do padrão “correto” é facilitada inmensuravelmente por um nome de padrão significativo.

Um template de padrões fornece um meio padronizado para descrição de padrões de projeto. Cada uma das entradas do template representa características do padrão de projeto que podem ser pesquisadas (por exemplo, por um banco de dados) e a maneira pela qual o padrão apropriado pode ser encontrado.

12.1.4 Linguagens e repositórios de padrões

Quando usamos o termo *linguagem*, a primeira coisa que nos vêm à mente é uma linguagem natural (por exemplo, inglês, espanhol, chinês) ou uma linguagem de programação (por exemplo, C++, Java). Em ambos os casos, a linguagem possui uma sintaxe e semântica usadas para transmitir ideias ou instruções procedurais de forma efetiva.

Quando se usa o termo *linguagem* no contexto de padrões de projeto, ele assume um significado ligeiramente diferente. Uma *linguagem de padrões* engloba um conjunto de padrões, cada qual descrito através do uso de um template de padrões (Seção 12.1.3) e inter-relacionados para mostrar como esses padrões colaboram para solucionar problemas em um domínio de aplicação.⁵

Em uma linguagem natural, as palavras são organizadas em sentenças que transmitem significado. A estrutura de sentenças é descrita pela sintaxe da linguagem. Em uma linguagem de padrões, os padrões de projeto são organizados para fornecer um “método estruturado para descrição de práticas de projeto adequadas em um domínio”.⁶

⁵ Christopher Alexander propôs originalmente linguagens de padrões para arquitetura e planejamento urbano. Hoje, as linguagens de padrões foram desenvolvidas para diversos setores, das ciências sociais ao processo de engenharia de software.

⁶ Essa descrição da Wikipedia pode ser encontrada em http://en.wikipedia.org/wiki/Pattern_language.



Caso não consiga encontrar uma linguagem de padrões que trate do domínio de seu problema, procure analogias em um outro conjunto de padrões.

WebRef

Para uma lista de linguagens de padrões úteis refira-se a c2.com/ppr/titles.html. Informações adicionais podem ser obtidas em hillside.net/patterns/.

De certa maneira, uma linguagem de padrões é análoga a um manual de instruções em hipertexto para resolução de problemas em áreas de aplicação específicas. O domínio do problema é descrito primeiro em termos hierárquicos, começando com problemas de projeto abrangentes associados ao domínio e então refinando-se cada um dos problemas abrangentes em níveis de abstração menores. No contexto de software, problemas de projeto abrangentes tendem a ser problemas de arquitetura por natureza e tratam a estrutura geral da aplicação e os dados ou conteúdo que o atendem. Os problemas de arquitetura são refinados para níveis de abstração menores, levando a padrões de projeto que resolvem subproblemas e colaboram entre si no nível de componentes (ou classes). Em vez de uma lista sequencial de padrões, a linguagem de padrões representa um conjunto interconectado em que o usuário pode partir de um problema de projeto abrangente e “ir fundo” para revelar problemas específicos e suas soluções.

Dezenas de linguagens de padrões foram propostas para projeto de software [Hil08]. Na maioria dos casos, os padrões de projeto que fazem parte da linguagem de padrões são armazenados em um repositório de padrões acessado via Web (por exemplo, [Boo08], [Cha03], [HPR02]). O repositório fornece um índice de todos os padrões de projeto e contém links de hipermídia que permitem ao usuário compreender as colaborações entre padrões.

12.2 PROJETO DE SOFTWARE BASEADO EM PADRÕES

Os melhores projetistas de qualquer área têm uma habilidade extraordinária de vislumbrar padrões que caracterizam um problema e padrões correspondentes que podem ser combinados para criar a solução. Os desenvolvedores de software da Microsoft [Mic04] discutem isso ao escreverem:

Embora o projeto baseado em padrões seja relativamente novo no campo de desenvolvimento de software, a tecnologia industrial tem usado projeto baseado em padrões há décadas, talvez há séculos. Catálogos de mecanismos e configurações padronizadas fornecem elementos de projeto que são usados para criar automóveis, aeronaves, máquinas-ferramenta e robôs. A aplicação de projeto baseado em padrões ao desenvolvimento de software promete os mesmos benefícios para o software como os já proporcionados à tecnologia industrial: previsibilidade, redução de riscos e maior produtividade.

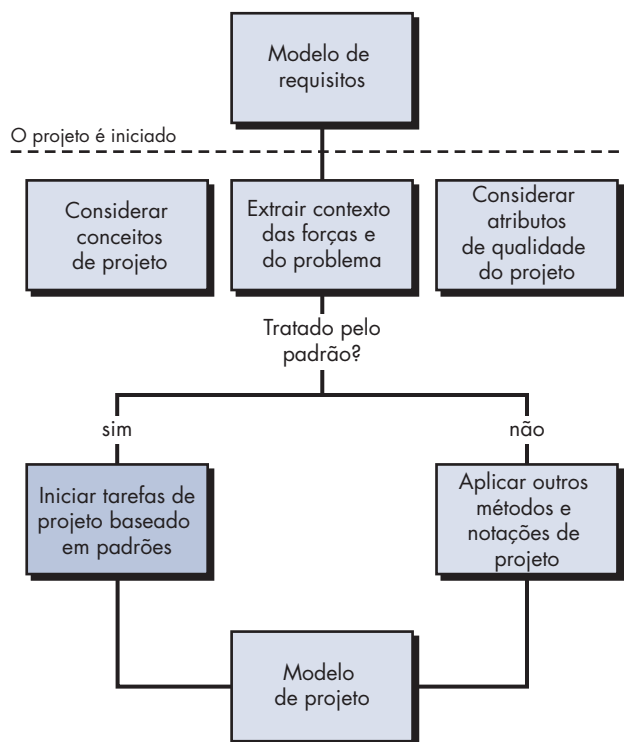
Ao longo do processo de projeto, devemos buscar toda oportunidade de aplicar padrões de projeto existentes (quando atenderem às necessidades do projeto) em vez de criar novos.

12.2.1 Contexto do projeto baseado em padrões

O projeto baseado em padrões não é utilizado isoladamente. Os conceitos e as técnicas discutidas para projeto da arquitetura, de componentes e para interfaces do usuário (Capítulos 9 a 11) são usados em conjunto com uma abordagem baseada em padrões. No Capítulo 8, citei que um conjunto de diretrizes e atributos de qualidade serve como base para todas as decisões de projeto de software. As próprias decisões são influenciadas por um conjunto de conceitos de projeto fundamentais (por exemplo, separação de preocupações, refinamento gradual, independência funcional) que são atingidos usando-se heurística que evoluiu ao longo de várias décadas e práticas melhores (por exemplo, técnicas, notação de modelagem) propostas para fazer com que o projeto seja mais fácil de ser realizado e mais efetivo como base para a construção.

O papel do projeto baseado em padrões está ilustrado na Figura 12.1. Um projetista de software inicia com um modelo de requisitos (seja ele explícito, seja implícito) que apresenta uma representação abstrata do sistema. O modelo de requisitos descreve o conjunto de problemas, estabelece o contexto e identifica o sistema de forças que exerce domínio. Talvez sugira o projeto de maneira abstrata, mas o modelo de requisitos faz pouco para representar o projeto explicitamente.

Ao iniciar seu trabalho como projetista, é sempre importante manter os atributos de qualidade em mente. Esses atributos (por exemplo, um projeto deve implementar todos os requisitos

FIGURA 12.1**Contexto do projeto baseado em padrões**

explícitos tratados no modelo de requisitos) estabelecem uma maneira de avaliar a qualidade do software, mas pouco fazem para ajudar a atingi-lo efetivamente. O projeto criado deve apresentar os conceitos fundamentais de projeto discutidos no Capítulo 8. Consequentemente, devemos aplicar técnicas comprovadas para traduzir as abstrações contidas no modelo de requisitos de maneira mais concreta que é o projeto de software. Para tanto, usaremos os métodos e as ferramentas de modelagem disponíveis para projeto da arquitetura, de componentes e para interfaces. Mas apenas quando depararmos com um problema, um contexto e um sistema de forças que ainda não foram resolvidos anteriormente. Se já existir uma solução, use-a! E isso significa aplicar uma abordagem de projeto baseado em padrões.

12.2.2 Pensando em termos de padrões

Em um excelente livro sobre projeto baseado em padrões, Shalloway e Trott [Sha05] comentam sobre uma “nova maneira de pensar” ao usarmos padrões como parte da atividade de projeto:

Eu tinha que me tornar mais receptivo a uma nova maneira de pensar. E quando fiz isso, ouvi [Christopher] Alexander dizer que “um bom projeto de software não pode ser alcançado simplesmente juntando peças operantes”.

Um bom projeto começa levando-se em conta o contexto — a visão geral. Quando o contexto é avaliado, extraímos uma hierarquia de problemas que devem ser resolvidos. Alguns desses problemas serão de natureza global, enquanto outros tratarão características e funções específicas do software. Todos serão afetados por um sistema de forças que irá influenciar a natureza de uma solução proposta.

Shalloway e Trott [Sha05] sugerem a seguinte abordagem⁷ que permite a um projetista pensar em termos de padrões:

7 Baseado no trabalho de Christopher Alexander [Ale79].

 O projeto baseado em padrões parece interessante para o problema que devo resolver. Por onde devo começar?

1. Certifique-se de ter entendido o quadro geral – o contexto em que o software a ser construído reside. O modelo de requisitos deve comunicar isso a você.
2. Examine o quadro geral, extraia os padrões presentes neste nível de abstração.
3. Inicie seu projeto com os padrões do “quadro geral” que estabeleçam um contexto ou esqueleto para trabalho de projeto posterior.
4. “Trabalhe em direção à essência, partindo do contexto” [Sha05] buscando padrões em níveis de abstração mais baixos que contribuam para a solução de projeto.
5. Repita os passos 1 a 4 até que o projeto completo ganhe corpo.
6. Refina o projeto adaptando cada padrão às especificidades do software que está tentando construir.

É importante notar que os padrões não são entidades independentes. Os padrões de projeto que estão presentes em nível de abstração elevado influenciarão invariavelmente a maneira pela qual outros padrões serão aplicados em níveis de abstração mais baixos. Além disso, os padrões em geral colaboram entre si. A implicação — ao selecionar um padrão de arquitetura, ele poderá influenciar os padrões de projeto escolhidos no nível de componentes. Da mesma forma, ao selecionar um padrão de projeto específico para interfaces, muitas vezes você se vê forçado a usar outros padrões que colaboram com ele.

Para ilustrarmos, consideremos a WebApp **CasaSeguraGarantida.com**. Se considerarmos o quadro geral, a WebApp deve tratar uma série de problemas fundamentais como:

- Como fornecer informações sobre os produtos e serviços *CasaSegura*.
- Como vender aos clientes os produtos e serviços *CasaSegura*.
- Como estabelecer monitoramento e controle baseado em Internet de um sistema de segurança instalado.

Cada um desses problemas fundamentais pode ser refinado ainda mais em um conjunto de subproblemas. Por exemplo, *Como vender via Internet* implica um padrão **E-commerce (para comércio eletrônico)** que por si só implica um grande número de padrões em níveis de abstração mais baixos. O padrão **E-commerce** (provavelmente, um padrão de arquitetura) implica mecanismos para configurar uma conta de cliente, exibir os produtos a ser vendidos, selecionar produtos para compra e assim por diante. Portanto, se pensarmos em padrões, é importante determinar se um padrão para configurar uma conta existe ou não. Se **ConfigurarConta** estiver disponível como um padrão viável para o contexto do problema, talvez ele colabore com outros padrões como **CriarFormulárioDeEntrada**, **GerenciarPreenchimentoDeFormulários** e **ValidarPreenchimentoFormulários**. Cada um desses padrões delinea problemas a ser resolvidos e soluções que possam ser aplicadas.

12.2.3 Tarefas de projeto

As tarefas de projeto a seguir são aplicadas quando se adota uma filosofia de projeto baseado em padrões:

1. Examine o modelo de requisitos e desenvolva uma hierarquia de problemas.

Descreva cada problema e subproblema isolando o problema, o contexto e o sistema de forças que se aplicam. Trabalhe inicialmente em problemas mais abrangentes (nível de abstração elevado) indo depois para subproblemas menores (em níveis de abstração mais baixos).

2. **Determine se uma linguagem de padrões confiável foi desenvolvida para o domínio do problema.** Conforme observado na Seção 12.1.4, uma linguagem de padrões trata problemas associados a um domínio de aplicação específico. A equipe de software do *CasaSegura* procuraria uma linguagem de padrões desenvolvida especificamente para produtos de segurança domiciliar. Se esse nível de especificidade de linguagem de padrões não puder ser encontrado, a equipe subdividiria o problema de software *CasaSegura* em uma série de

 Quais são as tarefas necessárias para criar um projeto baseado em padrões?

domínios de problemas genéricos (por exemplo, problemas de monitoramento de dispositivos digitais, problemas de interfaces do usuário, problemas de gerenciamento de vídeo digital) e buscaria linguagens de padrões apropriadas.

3. **Inicie com um problema abrangente, determine se um ou mais padrões de arquitetura se encontram disponíveis para ele.** Se existir um padrão de arquitetura, certifique-se de examinar todos os padrões colaboradores. Se o padrão for apropriado, adapte a solução de projeto proposta e construa um elemento de modelo de projeto que o represente adequadamente. Conforme observado na Seção 12.2.2, um problema abrangente para a Web-App **CasaSeguraGarantida.com** é tratada com um padrão **E-commerce** (de comércio eletrônico). Este sugerirá uma arquitetura específica para lidar com os requisitos de comércio eletrônico.
4. **Use as colaborações fornecidas para o padrão de arquitetura, examine problemas de subsistemas ou de componentes e procure padrões apropriados para tratar deles.** Talvez seja necessário pesquisar outros repositórios de padrões, bem como a lista de padrões que corresponde à solução de arquitetura. Se for encontrado um padrão apropriado, adapte a solução de projeto proposta e construa um elemento de modelo de projeto que o represente adequadamente. Certifique-se de aplicar o passo 7.
5. **Repita os passos 2 a 5 até que todos os problemas mais amplos tenham sido tratados.** A implicação é começar com o quadro geral e elaborar para resolver problemas em níveis cada vez mais detalhados.
6. **Se os problemas de projeto para interfaces do usuário tiverem sido isolados (quase sempre esse é o caso), pesquise os vários repositórios de padrões de projeto para interfaces do usuário em busca de padrões apropriados.** Prossiga de maneira similar para os passos 3, 4 e 5.
7. **Independentemente de seu nível de abstração, se uma linguagem de padrões e/ou repositório de padrões ou padrão individual se mostrarem promissores, compare o problema a ser resolvido em relação ao(s) padrão(ões) existente(s) apresentado(s).** Certifique-se de examinar o contexto e as forças para garantir que o padrão forneça, de fato, uma solução suscetível para o problema.
8. **Certifique-se de refinar o projeto à medida que é obtido de padrões usando critérios de qualidade de projeto como guia.**

Embora essa abordagem de projeto seja *top-down*, as soluções de projeto na vida real são, algumas vezes, mais complexas. Gillis [Gil06] tece comentários sobre isso ao escrever:

Os padrões de projeto na engenharia de software destinam-se ao uso racional e dedutivo. Portanto, temos o problema ou requisito geral, X, o padrão de projeto Y resolve X, consequentemente usamos Y. Agora, ao refletir sobre meu próprio processo — e tenho razões para acreditar que não sou o único a pensar assim — descubro que é mais orgânico que isso, mais indutivo que dedutivo, mais *bottom-up* que *top-down*. Obviamente, há um equilíbrio a ser atingido. Quando um projeto se encontra na fase inicial de partida e estou tentando passar dos requisitos abstratos para uma solução de projeto concreta, em geral realizarei uma busca inicial de grande amplitude... Achei que os padrões de projeto são úteis, permitindo-me rapidamente formular o problema de projeto em termos concretos.

Além disso, a abordagem baseada em padrões deve ser usada em conjunto com outros conceitos e técnicas de projeto de software.



As entradas na tabela podem ser complementadas com uma indicação da aplicabilidade relativa do padrão.

12.2.4 Construção de uma tabela para organização de padrões

À medida que o projeto baseado em padrões prossegue, talvez encontremos problemas para organizar e classificar prováveis padrões contidos em vários repositórios e linguagens de padrões. Para auxiliar a organizar nossa avaliação de prováveis padrões, a Microsoft [Mic04] sugere a criação de uma *tabela para organização de padrões* que assume a forma geral indicada na Figura 12.2.

FIGURA 12.2

Uma tabela para organização de padrões

Fonte: Adaptado de (Mic04)

	Banco de dados	Aplicação	Implementação	Infraestrutura
Dados/Conteúdo				
Enunciado do problema...	Nome(s)do(s)Padrão(ões)		Nome(s)do(s)Padrão(ões)	
Enunciado do problema...		Nome(s)do(s)Padrão(ões)		Nome(s)do(s)Padrão(ões)
Enunciado do Problema...	Nome(s)do(s)Padrão(ões)			Nome(s)do(s)Padrão(ões)
Arquitetura				
Enunciado do problema...		Nome(s)do(s)Padrão(ões)		
Enunciado do problema...		Nome(s)do(s)Padrão(ões)		Nome(s)do(s)Padrão(ões)
Enunciado do problema...				
Componentes				
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Enunciado do problema...				Nome(s)do(s)Padrão(ões)
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Interface do usuário				
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	

Uma tabela para organização de padrões pode ser implementada como um modelo de planilha usando o formato mostrado na figura. Uma lista resumida dos enunciados dos problemas, organizados por tópicos como dados/conteúdo, arquitetura, componentes e interfaces do usuário, é apresentada na coluna mais à esquerda (sobra mais escura). Quatro tipos padrão — bancos de dados, aplicação, implementação e infraestrutura — são listados ao longo da linha superior. Os nomes de prováveis padrões são indicados nas células da tabela.

Para fornecermos entradas para a tabela organizadora, pesquisaremos várias linguagens e repositórios de padrões em busca de padrões que atendam determinado enunciado de problema. Quando forem encontrados um ou mais prováveis padrões, estes são introduzidos na linha correspondente ao enunciado do problema e a coluna correspondente ao tipo de padrão. O nome do padrão é introduzido como um hiperlink para o URL do endereço Web que contém uma descrição completa do padrão.

12.2.5 Erros comuns de projeto

O projeto baseado em padrões pode lhe tornar um melhor projetista de software, mas ele não é uma panaceia. Como qualquer método de projeto, temos de começar com os primeiros princípios, enfatizando os fundamentos da qualidade de software e garantindo que o projeto atenda, de fato, às necessidades expressas pelo modelo de requisitos.

Uma série de erros comuns ocorre quando se usa projeto baseado em padrões. Em alguns casos, não se dedicou tempo suficiente para entender o problema subjacente e seu contexto e forças e, como consequência, escolhe-se um padrão que parece adequado, porém, é inadequado para a solução exigida. Assim que o padrão incorreto é escolhido, nos recusamos a admitir o erro e forçamos a adequação do padrão. Em outros casos, o problema possui forças não consideradas pelo padrão escolhido, resultando uma adequação deficiente ou errônea.

Algumas vezes um padrão é aplicado de forma demasiadamente literal e as adaptações necessárias para o espaço do problema não são implementadas. Esses erros poderiam ser evita-



Não force um padrão, mesmo que atenda ao problema em questão. Se o contexto e as forças estiverem errados, procure outro padrão.

dos? Na maioria dos casos a resposta é “sim”. Todo bom projetista pede uma segunda opinião e aceita revisões em seu trabalho. As técnicas de revisão discutidas no Capítulo 15 podem ajudar a garantir que o projeto baseado em padrões que desenvolvemos resultará em uma solução de alta qualidade para o problema de software a ser solucionado.

12.3 PADRÕES DE ARQUITETURA

PONTO-CHAVE

Uma arquitetura de software pode ter uma série de padrões de arquitetura que tratam questões como concorrência, persistência e distribuição.

 Quais os domínios de padrões de arquitetura típicos?

Se um engenheiro civil decide construir uma casa colonial com hall central, existe um único estilo arquitetônico a ser aplicado. Os detalhes do estilo (por exemplo, número de lareiras, fachada da casa, posição das portas e janelas) podem variar consideravelmente, mas uma vez tomada a decisão sobre a arquitetura geral da casa, o estilo é imposto sobre o projeto.⁸

Os padrões de arquitetura são um pouco diferentes. Por exemplo, toda casa (e todo estilo arquitetônico para casas) usa um padrão **Kitchen** (cozinha). O padrão **Kitchen** e padrões com os quais ele colabora atendem problemas associados ao armazenamento e à preparação de alimentos, os utensílios necessários para cumprir essas tarefas e regras para a colocação dos utensílios em relação ao fluxo no ambiente. Além disso, o padrão poderia atender problemas associados a balcões, iluminação, interruptores, uma ilha central, pisos e assim por diante. Obviamente, há mais de um projeto para uma cozinha, em geral ditado pelo contexto e sistema de forças. Mas todo projeto pode ser concebido no contexto da “solução” sugerida pelo padrão **Kitchen**.

Conforme citado, os padrões de arquitetura para software definem uma abordagem específica para tratar alguma característica do sistema. Bosch [Bos00] e Booch [Boo08] definem uma série de domínios de padrões de arquitetura. Exemplos representativos são apresentados nos parágrafos seguintes:

Controle de acesso. Há várias situações em que o acesso a dados, recursos e funcionalidade fornecidos por uma aplicação é limitado a usuários finais especificamente definidos. Do ponto de vista da arquitetura, o acesso a alguma parte da arquitetura de software deve ser rigorosamente controlado.

Concorrência. Muitas aplicações têm de tratar múltiplas tarefas em um modo que simule paralelismo (isso ocorre sempre que vários componentes ou tarefas “paralelos” são administrados por um único processador). Há uma série de maneiras diferentes com a qual uma aplicação pode tratar a concorrência, e cada uma delas pode ser apresentada por um padrão de arquitetura distinto. Por exemplo, uma abordagem é usar um padrão **Operating-System-ProcessManagement** (sistema operacional-gerenciamento de processos) que fornece recursos embutidos no sistema operacional que permitem aos componentes executarem de forma concorrente. O padrão também incorpora funcionalidade que gerencia a comunicação entre processos, agendamento e outras capacidades exigidas para alcançar a concorrência. Uma outra abordagem poderia ser definir um agendador de tarefas no nível da aplicação. Um padrão **TaskScheduler** (agendador de tarefas) contém um conjunto de objetos ativos em que cada um contém uma operação *tick()* [Bos00]. O agendador chama periodicamente *tick()* para cada objeto, que então realiza as funções que ele deve realizar antes de retornar o controle de volta para o agendador, que então chama a operação *tick()* para o próximo objeto concorrente.

Distribuição. O problema da distribuição trata a maneira pela qual os sistemas ou componentes nos sistemas se comunicam entre si em um ambiente distribuído. São considerados dois subproblemas: (1) a maneira pela qual as entidades se conectam entre si e (2) a natureza

⁸ Isso implica que haverá um *hall* central e um corredor, que as salas serão colocadas à esquerda e à direita do *hall*, que a casa terá dois (ou mais) pisos, que os quartos da casa estarão no piso de cima e assim por diante. Essas “regras” são impostas uma vez tomada a decisão de adotar o estilo colonial com *hall* central.

da comunicação que ocorre. O padrão de arquitetura mais comum estabelecido para tratar o problema de distribuição é o padrão **Broker** (agente). Um agente atua com um “intermediário” entre o componente-cliente e o componente-servidor. O cliente envia uma mensagem ao agente (contendo todas as informações apropriadas para a comunicação a ser efetuada) e o agente completa a conexão.

Persistência. Os dados persistem se sobreviverem depois da execução do processo que o criou. Os dados persistentes armazenados em um banco de dados ou arquivo podem ser lidos ou modificados por outros processos posteriormente. Em ambientes orientados a objetos, a ideia de um objeto persistente estende um pouco mais o conceito de persistência. Os valores de todos os atributos do objeto, o estado geral do objeto e outras informações complementares são armazenados para recuperação e uso futuros. Em geral, usam-se dois padrões de arquitetura para obter a persistência — o padrão **DatabaseManagementSystem** (sistema de gerenciamento de bancos de dados), que aplica o recurso de armazenamento e recuperação de um DBMS à arquitetura da aplicação, ou o padrão **Application Level-Persistence** (persistência no nível de aplicação) que constrói recursos de persistência na arquitetura da aplicação (por exemplo, software de processamento de texto que gerencia sua própria estrutura de documentos).

Antes que qualquer um dos padrões de arquitetura representativos citados nos parágrafos anteriores possa ser escolhido, ele deve ser avaliado em termos de sua adequação para a aplicação e estilo de arquitetura geral, bem como o contexto e o sistema de forças que ele especifica.



Repositórios de padrões de projeto

Existem várias fontes de padrões de projeto disponíveis na Web. Alguns padrões podem ser obtidos de linguagens de padrões publicadas individualmente, enquanto outros se encontram disponíveis como parte de um portal ou repositório de padrões. Vale a pena dar uma olhada nas seguintes fontes na Web:

Hillside.net <http://hillside.net/patterns/> Um dos conjuntos mais completos da Web em termos de padrões e linguagens de padrões.

Portland Pattern Repository

<http://c2.com/ppr/index.html>

Contém links para uma ampla gama de recursos e coleções de padrões.

Pattern Index <http://c2.com/cgi/wiki?PatternIndex>

Uma “coleção de padrões eclética”

Booch’s Architecture Patterns Handbook

www.booch.com/architecture/index.jsp

Referência bibliográfica para centenas de padrões de projeto de arquitetura e componentes

Coleções de padrões para interfaces do usuário

Padrões para UI/HCI

www.hcipatterns.org/patterns.html

Jennifer Tidwell’s UI patterns

www.time-tripper.com/uipatterns/

Padrões de projeto para interfaces do usuário móveis

<http://patterns.littlespringsdesign.com/wikka.php?wakka=Mobile>

Padrões

Linguagem de padrões para projeto de interfaces do usuário

www.maplefish.com/todd/papers/Experiences.html

Biblioteca de projeto interativo para jogos

www.eelke.com/research/usability.html

Padrões de projeto para interfaces do usuário

www.cs.helsinki.fi/u/salaakso/patterns/

Padrões de projeto especializados:

Aviônica

<http://g.oswego.edu/dl/acs/acs/acs.html>

Sistemas de informações de negócios

www.objectarchitects.de/arcus/cookbook/

Processamento distribuído

www.cs.wustl.edu/~schmidt/

Padrões IBM para e-Business

www128.ibm.com/developerworks/patterns/

Biblioteca de padrões de projeto do Yahoo!

<http://developer.yahoo.com/ypatterns/>

WebPatterns.org <http://webpatterns.org/>

INFORMAÇÕES

12.4 PADRÕES DE PROJETO DE COMPONENTES

Os padrões de projeto de componentes nos dão soluções comprovadas que tratam um ou mais subproblemas extraídos do modelo de requisitos. Em muitos casos, os padrões de projeto desse tipo se concentram em algum elemento funcional de um sistema. Por exemplo, a aplicação **CasaSeguraGarantida.com** deve tratar o seguinte subproblema de projeto: *Como podemos obter especificações de produtos e informações relacionadas para qualquer dispositivo do CasaSegura?*

Tendo enunciado o subproblema que deve ser resolvido, devemos considerar agora o contexto e o sistema de forças que afetam uma solução. Examinando-se o caso de uso de modelo de requisitos apropriado, constataremos que o consumidor usa a especificação para um dispositivo do *CasaSegura* (por exemplo, um sensor ou câmera de segurança) para fins de informação. Entretanto, outras informações relacionadas à especificação (por exemplo, determinação de preços) poderiam ser empregadas quando a funcionalidade de comércio eletrônico fosse selecionada.

A solução para o subproblema envolve uma pesquisa. Como pesquisar é um problema muito comum, não deve ser nenhuma surpresa a existência de muitos padrões relacionados à pesquisa. Pesquisando uma série de repositórios de padrões, descobriremos os seguintes padrões, juntamente com o problema que cada um resolve:

AdvancedSearch. Os usuários precisam encontrar um item específico em um grande conjunto de itens.

HelpWizard. Os usuários precisam de ajuda sobre certo tópico relativo ao site ou quando eles precisam encontrar uma página específica dentro desse site.

SearchArea. Os usuários precisam encontrar uma página Web.

SearchTips. Os usuários precisam saber como controlar o mecanismo de busca.

SearchResults. Os usuários têm de processar uma lista de resultados de uma busca.

SearchBox. Os usuários têm de encontrar um item ou informações específicas.

Para **CasaSeguraGarantida.com** o número de produtos não é particularmente grande e cada um deles possui uma classificação relativamente simples, de modo que **AdvancedSearch** e **HelpWizard** provavelmente não sejam necessários. Similarmente, a busca é relativamente simples para não precisar de **SearchTips**. A descrição de **SearchBox**, entretanto, é dada (em parte) como:

Search Box

(Adaptado de www.welie.com/patterns/showPattern.php?patternID=search.)

Problema: Os usuários precisam encontrar um item ou informações específicos.

Motivação: Qualquer situação em que uma busca com palavra-chave é aplicada em um conjunto de objetos de conteúdo organizados na forma de páginas Web.

Contexto: Em vez de usar navegação para obter informações ou conteúdo, o usuário quer fazer uma busca direta em conteúdo contido em várias páginas Web. Qualquer site que já possui recursos primários de navegação. O usuário poderia querer procurar um item numa categoria. Ele poderia querer especificar uma consulta de forma mais detalhada.

Forças: O site já possui recursos primários de navegação. Os usuários talvez queiram procurar um item de determinada categoria. Eles poderiam querer especificar uma consulta, de forma mais detalhada, através de operadores booleanos simples.

Solução: Oferecer funcionalidade de busca formada por um identificador de busca, um campo para palavra-chave, um filtro se aplicável e um botão “avançar”. Pressionar a tecla Enter tem a mesma função que selecionar o botão Avançar. Também fornece Dicas de Busca e exemplos em uma página separada. Um link para essa página é colocado próximo à funcionalidade de busca. A caixa de edição para o termo de pesquisa é suficientemente grande para acomodar três consultas de usuário típicas (normalmente, por volta de 20 caracteres). Se o número de filtros for maior que 2, use uma caixa de combinação para seleção de filtros ou um botão de rádio.

Os resultados da pesquisa são apresentados em uma nova página com um identificador claro contendo pelo menos “Resultados da busca” ou algo similar. A função de busca é repetida na parte superior da página com as palavras-chave anteriormente introduzidas, de modo que os usuários saibam quais eram elas.

A descrição de padrões continua com outras entradas conforme descrito na Seção 12.1.3.

O padrão prossegue para descrever como os resultados da busca são acessados, apresentados, correspondidos e assim por diante. Baseado nisso, a equipe do **CasaSeguraGarantida.com** pode projetar os componentes necessários para implementar a busca ou (mais provavelmente) obter componentes reutilizáveis existentes.

CASASEGURA



Aplicação de padrões

Cena: Discussão informal durante o projeto de um incremento do software que implementa controle de sensores via Internet para o **CasaSegura-Garantida.com**.

Atores: Jamie (responsável pelo projeto) e Vinod (arquiteto-chefe do sistema **CasaSeguraGarantida.com**).

Conversa:

Vinod: Então, como está indo o projeto da interface para controle de câmeras?

Jamie: Nada mal. Já desenhei a maior parte da capacidade de conexão com os verdadeiros sensores sem grandes problemas. Já comecei também a pensar sobre a interface para os usuários poderem efetivamente movimentar, deslocar e ampliar as câmeras de uma página Web remota, porém ainda não estou certo de tê-la feito corretamente.

Vinod: Qual sua ideia?

Jamie: Bem, entre os requisitos, o controle de câmeras precisa ser altamente interativo — à medida que o usuário movimenta o controle, a câmera deve se movimentar o quanto antes possível. Portanto, estava imaginando um conjunto de botões dispostos como uma câmera comum, porém, ao clicá-los, o usuário controlaria a câmera.

Vinod: Hummm. Sim, isso funcionaria, mas não estou certo de que esteja correto — para cada clique de um controle precisaríamos esperar que ocorresse toda a comunicação cliente/servidor e, portanto, não teríamos a sensação de uma resposta imediata.

Jamie: Foi isso que imaginei — pelo fato de não estar muito satisfeito com a abordagem, mas não sei exatamente como poderia fazê-lo de outra forma.

Vinod: Bem, por que não simplesmente usa o padrão **InteractiveDeviceControl**?!

Jamie: Hummm — o que é isso? Creio nunca ter ouvido falar dele.

Vinod: Basicamente trata-se de um padrão destinado exatamente para o problema que você está me descrevendo. A solução que ele propõe é criar uma conexão de controle do servidor com o dispositivo, por meio da qual comandos de controle poderiam ser enviados. Dessa maneira, não seria preciso enviar solicitações HTTP normais. E o padrão ainda indicaria como implementar isso usando algumas técnicas AJAX simples. Temos alguns JavaScripts simples no cliente que se comunicam diretamente com o servidor e transmitem os comandos assim que o usuário fizer alguma coisa.

Jamie: Legal! Era exatamente isso que eu precisava para resolver essa questão. Onde posso encontrá-lo?

Vinod: Ele se encontra à disposição em um repositório on-line. Eis a URL.

Jamie: Vou verificar isso.

Vinod: Sim — mas lembre-se de verificar as consequências para o padrão. Parece que eu me lembro de haver algo lá sobre precisar tomar cuidado com questões de segurança. Imagino que possa ser devido à criação de um canal de controle separado e, portanto, evitando os mecanismos de segurança comuns da Web.

Jamie: Excelente observação. Provavelmente não teria atentado a esse fato! Obrigado.

12.5 PADRÕES DE PROJETO PARA INTERFACES DO USUÁRIO

Foram propostas centenas de padrões para interfaces do usuário (UI, em inglês) nos últimos anos. A maior parte deles cai em uma das seguintes dez categorias de padrões (discutidas com um exemplo representativo⁹) conforme descrito por Tidwell [Tid02] e vanWelie [Wel01]:

Toda a Interface com o Usuário. Fornece orientação para estrutura de alto nível e navegação por toda a interface.

Padrão: TopLevelNavigation

Descrição: Usada quando um site ou uma aplicação implementa uma série de funções principais. Oferece um menu de alto nível, geralmente acoplado a um logo ou imagem identificadora, que possibilita a navegação direta para qualquer uma das principais funções do sistema.

Detalhes: Funções principais (em geral limitadas a quatro a sete nomes de função) são listadas na parte superior da tela (possível também formatos de colunas verticais) em uma linha horizontal de texto. Cada nome fornece um link para uma fonte de informações ou função apropriada. Geralmente, usada com o padrão **BreadCrumbs** discutido mais à frente.

Elementos de navegação: Cada nome de função/conteúdo representa um link para a função ou conteúdo apropriado.

Layout da página. Trata a organização geral das páginas (para sites) ou exibições em tela distintas (para aplicações interativas).

Padrão: CardStack

Descrição: Usado quando uma série de subfunções ou categorias de conteúdo específicas relacionadas com um recurso ou função deve ser selecionada em ordem aleatória. Dá a aparência de uma pilha de fichas indexadoras, cada uma delas selecionável com um clique de mouse e cada qual representando subfunções ou categorias de conteúdo específicas.

Detalhes: As fichas indexadoras são uma metáfora bem compreendida e fácil para o usuário manipular. Cada ficha indexadora (separador) pode ter um formato ligeiramente diverso. Alguns poderão exigir entrada de dados e possuir botões ou outros mecanismos de navegação; outros podem ser informativos. Poderiam ser combinados com outros padrões como **DropDownList**, **Fill-in-the-Blanks** e outros.

Elementos de navegação: Um clique de mouse em uma guia faz com que a ficha apropriada apareça. Os recursos de navegação contidos na ficha também poderiam estar presentes, porém, em geral, estas deveriam iniciar uma função relacionada aos dados da ficha e não provocar um link real para alguma outra tela.

Formulários e introdução de dados. Considera uma grande variedade de técnicas de projeto para realizar a introdução de dados em formulários.

Padrão: Fill-in-the-Blanks

Descrição: Possibilita que dados alfanuméricos sejam introduzidos em uma “caixa de texto.”

Detalhes: Os dados poderiam ser introduzidos em uma caixa de texto. Em geral, são validados e processados após a seleção de algum indicador de texto ou gráfico (por exemplo, um botão contendo “avançar”, “submeter”, “próximo”). Em muitos casos, esse padrão pode ser combinado com uma lista suspensa ou outros padrões (por exemplo, *SEARCH <drop down list> FOR <fill-in-the-blanks text box>*).

Elementos de navegação: Um indicador de texto ou gráfico que inicia a validação e o processamento.

Tabelas. Fornece orientação de projeto para a criação e manipulação de dados tabulares de toda espécie.

⁹ Um template-padrão sintetizado é usado aqui. Descrições de padrões completas (juntamente com dezenas de outros padrões) podem ser encontradas em [Tid02] e [Wel01].

Padrão: SortableTable

Descrição: Mostra uma longa lista de registros que podem ser ordenados através da seleção de um mecanismo comutador para qualquer rótulo de coluna.

Detalhes: Cada linha da tabela representa um registro completo. Cada coluna representa um campo no registro. Cada título de coluna é, na verdade, um botão selecionável que pode ser comutado para iniciar uma ordem crescente ou decrescente no campo associado à coluna para todos os registros exibidos. Em geral a tabela é redimensionável e poderá ter um mecanismo de rolagem caso o número de registros seja maior que o espaço de janela disponível.

Elementos de navegação: Cada cabeçalho de coluna inicia uma classificação de todos os registros. Nenhuma outra navegação é fornecida, embora em alguns casos, cada registro poderia por si só conter links de navegação para outro conteúdo ou funcionalidade.

Manipulação de dados diretos. Lida com a edição, a modificação e a transformação de dados.

Padrão: BreadCrumbs

Descrição: Fornece um caminho de navegação completo quando o usuário está trabalhando com uma hierarquia de páginas ou telas complexa.

Detalhes: É atribuído um identificador exclusivo a cada página ou tela. O caminho de navegação para o local atual é especificado em um local predefinido para qualquer tela. O caminho assume a forma: **homepage>página de tópico principal>página de subtópico>página específica>página atual.**

Elementos de navegação: Qualquer uma das entradas contidas na tela Bread Crumbs pode ser usada como um ponteiro para um link de retorno para um nível hierárquico mais elevado.

Navegação. Ajuda o usuário na navegação através de menus hierárquicos, páginas Web e telas interativas.

Padrão: EditInPlace

Descrição: Fornece um recurso de edição simples para certos tipos de conteúdo no local em que é exibido. Nenhuma necessidade de o usuário introduzir explicitamente uma função ou modo de edição de texto.

Detalhes: O usuário vê o conteúdo na tela que deve ser alterado. Um clique duplo com o mouse sobre o conteúdo indica ao sistema que se deseja editar. O conteúdo é realçado para significar que o modo de edição está disponível e o usuário faz as mudanças apropriadas.

Elementos de navegação: Nenhum.

Searching. Possibilita buscas de conteúdo específicas nas informações mantidas em um site ou contidas em repositórios de dados persistentes que podem ser acessados via aplicação interativa.

Padrão: SimpleSearch

Descrição: Oferece a capacidade de pesquisar em um site ou fonte de dados persistentes para um dado simples descrito em uma string alfanumérica.

Detalhes: Oferece a capacidade de pesquisar local (uma página ou um arquivo) ou globalmente (o site todo ou um banco de dados completo) por uma string de busca. Gera uma lista de “acertos” em ordem de probabilidade para atender às necessidades do usuário. Não oferece buscas de itens múltiplos ou operações booleanas especiais (veja *padrão de busca avançado*).

Elementos de navegação: Cada entrada na lista de acertos representa um link de navegação com os dados referidos pela entrada.

Elementos de página. Implementa elementos específicos de uma página Web ou tela.

Padrão: Wizard

Descrição: Conduz o usuário, passo a passo, através de uma tarefa complexa, dando orientação para a finalização da tarefa por meio de uma série de telas de janelas simples.

Detalhes: O exemplo clássico é um processo de registro em quatro etapas. O padrão Wizard gera uma janela para cada etapa, solicitando informações específicas do usuário, um passo por vez.

Elementos de navegação: Navegação para a frente e para trás que permite ao usuário revisitar cada passo dado no processo de assistente.

E-commerce. Específicos para sites, esses padrões implementam elementos recorrentes de aplicações para comércio eletrônico.

Padrão: ShoppingCart

Descrição: Fornece uma lista de itens selecionados para compra.

Detalhes: Lista informações de itens, quantidade, código de produto, disponibilidade (disponível, esgotado), preço, informações para entrega, custos de remessa e outras informações de compra relevantes. Também oferece a habilidade de editar (por exemplo, remover, modificar quantidade).

Elementos de navegação: Contém a capacidade de prosseguir com a compra ou sair para encerrar as compras.

Miscellaneous. Padrões que não se encaixam perfeitamente em uma das categorias anteriores. Em alguns casos, dependem do domínio ou ocorrem apenas para classes de usuário específicas.

Padrão: ProgressIndicator

Descrição: Dá uma indicação do progresso quando uma operação leva mais do que n segundos.

Detalhes: Representado na forma de um ícone de animação ou uma caixa de mensagens contendo alguma indicação visual (por exemplo, um “poste de barbeiro” girante, um controle deslizante com um indicador da porcentagem completada) de que o processamento está em andamento. Também pode conter uma indicação de conteúdo de texto do estado do processamento.

Elementos de navegação: Em geral contém um botão que permite ao usuário pausar ou cancelar processamento.

Cada um dos exemplos de padrões anteriores (e todos os padrões em cada categoria) também teria um projeto completo de componentes, incluindo classes de projeto, atributos, operações e interfaces.

Uma discussão completa para interfaces do usuário vai além do escopo deste livro. Caso tenha maior interesse, veja [Duy02], [Bor01], [Tid02] e [Wel01] para mais informações.

12.6 PADRÕES DE PROJETO PARA WEBAPPS

Ao longo deste capítulo vimos que há diferentes tipos de padrões e várias maneiras distintas em que podem ser classificados. Ao considerarmos os problemas de projeto a ser solucionados quando uma WebApp é construída, vale a pena considerar categorias de padrão concentrando-nos em duas dimensões: o foco de projeto do padrão e seu nível de granularidade. O *foco do projeto* identifica qual aspecto do modelo de projeto é relevante (por exemplo, arquitetura das informações, navegação, interação). A *granularidade* identifica o nível de abstração que está sendo considerado (por exemplo, o padrão se aplica a toda a WebApp ou a uma única página Web, a um subsistema ou um componente WebApp individual?).

12.6.1 Foco do projeto

Em capítulos anteriores enfatizei uma progressão de projeto que inicia considerando-se questões de arquitetura e de componentes e representações de interfaces do usuário. A cada etapa, os problemas considerados e as soluções propostas começam com um alto nível de abstração e lentamente se tornam mais detalhados e específicos. Em outras palavras, o foco do

PONTO-CHAVE

O foco se torna “mais limitado” quanto mais se avança no projeto.

projeto se torna “mais limitado” à medida que avançamos no projeto. Os problemas (e soluções) que encontraremos ao projetarmos uma arquitetura de informações para uma WebApp são diferentes dos problemas (e soluções) encontrados ao realizarmos o projeto de interfaces. Consequentemente, não será nenhuma surpresa que os padrões para o projeto de WebApps possa ser desenvolvido para diferentes níveis de foco de projeto, para podermos tratar os problemas únicos (e soluções relativas) encontrados em cada nível. Os padrões para WebApps podem ser classificados usando-se os seguintes níveis de foco do projeto:

- **Padrões de arquitetura de informações** relacionam-se à estrutura geral do espaço de informações e as maneiras pelas quais os usuários irão interagir com as informações.
- **Padrões de navegação** definem estruturas de links de navegação como hierarquias, anéis, *tours* e assim por diante.
- **Padrões de interação** contribuem para o projeto da interface do usuário. Os padrões nessa categoria tratam a maneira pela qual a interface informa o usuário das consequências de uma ação específica, como um usuário expande conteúdo baseado no contexto de uso e nos desejos dos usuários, como descrever melhor o destino definido por um link, como informar o usuário sobre o estado de uma interação em andamento e questões relacionadas com interfaces.
- **Padrões de apresentação** ajudam na apresentação do conteúdo à medida que é apresentado ao usuário via interface. Padrões nessa categoria tratam como organizar as funções de controle de interfaces do usuário para melhor usabilidade, como mostrar a relação entre uma ação de interface e os objetos de conteúdo que ela afeta e como estabelecer hierarquias de conteúdo efetivas.
- **Padrões funcionais** definem os fluxos de trabalho, comportamentos, processamento, comunicação e outros elementos algorítmicos contidos em uma WebApp.

Na maioria dos casos, seria infrutífero explorar o conjunto dos padrões de arquitetura de informações quando se encontra um problema no projeto de interação. Teríamos de examinar padrões de interação, pois esse é o foco do projeto relevante ao trabalho que está sendo realizado.

12.6.2 Granularidade do projeto

Quando um problema envolve questões de “quadro geral”, devemos tentar desenvolver soluções (e usar padrões relevantes) que se concentram no quadro geral. Quando o foco é muito limitado (por exemplo, selecionando unicamente um item de um pequeno conjunto de cinco ou menos itens), a solução (e o padrão correspondente) é atingida com pouca margem de erro. Em termos do nível de granularidade, os padrões podem ser descritos nos seguintes níveis:

- **Padrões de arquitetura.** Esse nível de abstração irá, tipicamente, se relacionar a padrões que definem a estrutura geral da WebApp, indicam os relacionamentos entre os diferentes componentes ou incrementos e definem as regras para especificar as relações entre os elementos (páginas, pacotes, componentes, subsistemas) da arquitetura.
- **Padrões de projeto.** Tratam um elemento específico do projeto como uma agregação de componentes para resolver algum problema de projeto, relações entre os elementos em uma página ou os mecanismos para efetuar a comunicação componente-para-componente. Um exemplo poderia ser o padrão **Broadsheet** para o layout da homepage de uma WebApp.
- **Padrões de componentes.** Esse nível de abstração relaciona-se a elementos individuais de pequena escala de uma WebApp. Entre os exemplos, temos elementos de interação individual (por exemplo, botões de rádio), itens de navegação (por exemplo, como formataríamos links?) ou elementos funcionais (por exemplo, algoritmos específicos). Também é possível definir a relevância de diferentes padrões para diferentes áreas ou categorias de aplicação. Por exemplo, um conjunto de padrões (em diferentes níveis da granularidade e foco do projeto) poderia ser particularmente relevante ao comércio eletrônico.

INFORMAÇÕES



Repositórios de padrões de projeto para hipermídia

O site IAWiki (<http://iawiki.net/WebsitePatterns>), um espaço de discussão colaborativa para arquitetos da informação, contém diversos recursos úteis. Entre eles temos links para uma série de catálogos de padrões e repositórios para hipermídia. Centenas de padrões de projeto estão representados:

Repositório de padrões de projeto para hipermídia

www.designpattern.lu.unisi.ch/

InteractionPatterns de Tom Erickson

www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html

Padrões de projeto para a Web de Martijn vanWelie

www.welie.com/patterns/

Padrões Web para projeto de interfaces do usuário

[http://harbinger.sims.berkeley.edu/](http://harbinger.sims.berkeley.edu/http://harbinger.sims.berkeley.edu/)

ui.designpatterns/webpatterns2/

webpatterns/home.php

Padrões para sites pessoais

www.rdrop.com/%7Ehalf/Creations/Writings/Web.patterns/index.html

Aperfeiçoamento de sistemas de informações na Web com padrões de navegação

<http://www8.org/w8-papers/>

5b-hypertext-media/improving/improving.html

Uma linguagem de padrões HTML 2.0

www.anamorph.com/docs/patterns/default.html

Pontos em comum — uma linguagem de padrões para projeto de HCIs

www.mit.edu/~jtidwell/interaction_patterns.html

Padrões para sites pessoais **www.rdrop.com/**

~half/Creations/Writings/Web.patterns/index.html

Indexação de linguagem de padrões **www.cs.brown.edu/~rms/InformationStructures/Indexing/**

~rms/InformationStructures/Indexing/Overview.html

Overview.html

12.7 RESUMO

Os padrões de projeto fornecem um mecanismo codificado para descrever problemas e suas soluções de maneira que permita à comunidade da engenharia de software capturar conhecimento de projeto para que possa ser reutilizado. Um padrão descreve um problema, indica o contexto, permitindo ao usuário compreender o ambiente em que o problema reside e listar um sistema de forças que indicam como o problema pode ser interpretado no seu contexto e como uma solução pode ser aplicada. No trabalho de engenharia de software, identificamos e documentamos padrões generativos que descrevem um aspecto importante e repetível de um sistema, dando-nos então uma forma de construir esse aspecto em um sistema de forças que seja único a um dado contexto.

Os padrões de arquitetura descrevem problemas de projeto abrangentes resolvidos usando-se uma abordagem estrutural. Os padrões de dados descrevem problemas recorrentes orientados a dados e as soluções de modelagem de dados que podem ser usadas para resolvê-las. Os padrões de componentes (também conhecidos como padrões de projeto) tratam de problemas associados ao desenvolvimento de subsistemas e componentes, a maneira pela qual eles se comunicam entre si e seu posicionamento em uma arquitetura mais ampla. Os padrões de projeto para interfaces descrevem problemas comuns de interfaces do usuário e suas soluções com um sistema de forças que inclui as características específicas dos usuários finais. Os padrões para WebApps lidam com um conjunto de problemas encontrado ao se construírem WebApps e muitas vezes incorporam muitas das demais categorias de padrões mencionados. Uma estrutura fornece a infraestrutura em que talvez possam residir padrões, e idiomas descrevem detalhes de implementação específicos a uma linguagem de programação para o todo ou parte de uma estrutura de dados ou algoritmo específico. Um formulário ou template-padrão é usado para descrições de padrões. Uma linguagem de padrões engloba um conjunto de padrões, cada qual descrito por meio do uso de um template padronizado e inter-relacionado para mostrar como os padrões colaboram para solucionar problemas em um campo de aplicação.

O projeto baseado em padrões é usado em conjunto com métodos de projeto para arquitetura, para a componentização e para interfaces do usuário. A abordagem de projeto começa com um exame do modelo de requisitos para isolar problemas, definir o contexto e descrever o siste-

ma de forças. Em seguida, buscam-se linguagens de padrões para o domínio do problema para determinar se há padrões para os problemas que foram isolados. Uma vez encontrados padrões apropriados, eles são usados como um guia de projeto.

PROBLEMAS E PONTOS A PONDERAR

- 12.1.** Discuta as três “partes” de um padrão de projeto e forneça um exemplo concreto de cada um deles de algum outro campo que não seja o de software.
- 12.2.** Qual a diferença entre um padrão não generativo e generativo?
- 12.3.** Como os padrões de arquitetura diferem dos padrões de componentes?
- 12.4.** O que é uma estrutura de uso de padrões e como ela difere de um padrão? O que é um idioma e como ele difere de um padrão?
- 12.5.** Usando o template de padrões de projeto apresentado na Seção 12.1.3, desenvolva uma descrição de padrão completa para um padrão sugerido pelo seu professor.
- 12.6.** Desenvolva uma linguagem de padrões estrutural para um esporte com o qual você esteja familiarizado. Você pode começar lidando com o contexto, o sistema de forças e os problemas abrangentes que um técnico e a equipe devem solucionar. É preciso não apenas especificar os nomes de padrão, mas também uma descrição em uma sentença para cada padrão.
- 12.7.** Encontre cinco repositórios de padrões e apresente uma descrição abreviada dos tipos de padrões contidos em cada um deles.
- 12.8.** Quando Christopher Alexander diz “um bom projeto de software não pode ser alcançado simplesmente juntando-se peças operantes”, o que você acha que ele quis dizer com isso?
- 12.9.** Usando as tarefas para projeto baseado em padrões citadas na Seção 12.2.3, desenvolva um esboço de projeto para o “sistema de design de interiores” descrito na Seção 11.3.2.
- 12.10.** Construa uma tabela para organizar padrões para os padrões utilizados no Problema 12.9.
- 12.11.** Usando o template de padrões de projeto apresentado na Seção 12.1.3, desenvolva uma descrição completa de padrões para o padrão **Kitchen** mencionado na Seção 12.3.
- 12.12.** A “gangue dos quatro” [Gam95] propôs uma variedade de padrões de componentes aplicáveis a sistemas orientados a objetos. Escolha um (estes se encontram à disposição na Web) e discuta-o.
- 12.13.** Encontre três repositórios de padrões para padrões de interfaces do usuário. Escolha um padrão de cada e apresente uma descrição abreviada dele.
- 12.14.** Encontre três repositórios de padrões para padrões de WebApps. Escolha um padrão de cada e apresente uma descrição abreviada dele.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Ao longo da última década, lançaram-se vários livros sobre projeto baseado em padrões destinados a engenheiros de software. Gamma e seus colegas [Gam95] escreveram o livro seminal sobre o tema. Entre algumas contribuições mais recentes, temos obras como as de Lasater (*Design Patterns*, Wordware Publishing, Inc., 2007), Holzner (*Design Patterns for Dummies*, For Dummies, 2006), Freeman e seus colegas (*Head First Design Patterns*, O'Reilly Media, Inc., 2005) e Shalloway e Trott (*Design Patterns Explained*, 2. ed., Addison-Wesley, 2004). Uma edição especial do *IEEE Software* (julho/agosto, 2007) discute uma variedade de tópicos de padrões de software. Kent Beck (*Implementation Patterns*, Addison-Wesley, 2008) trata de padrões para problemas de codificação e implementação encontrados durante a atividade de construção.

Outros livros focalizam padrões de projeto à medida que são fornecidos em ambientes de linguagem e desenvolvimento de aplicações específicos. Entre as contribuições nessa área, temos: Bowers (*Pro CSS and HTML Design Patterns*, Apress, 2007), Tropashko e Burleson (*SQL Design Patterns: Expert Guide to SQL Programming*, Rampant Techpress, 2007), Mahemoff (*Ajax Design Patterns*, O'Reilly Media, Inc., 2006), Metsker e Wake (*Design Patterns in Java*, Addison-Wesley, 2006), Nilsson (*Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*, Addison-Wesley, 2006), Sweat (*PHPArchitect's Guide to PHP Design Patterns*, Marco Tabini & Associates, Inc., 2005), Metsker (*Design Patterns C#*, Addison-Wesley, 2004), Grand e Merrill (*Visual Basic .NET Design Patterns*, Wiley, 2003), Crawford e Kaplan (*J2EE Design Patterns*, O'Reilly Media, Inc., 2003), Juric et al. (*J2EE Design Patterns Applied*, Wrox Press, 2002) e Marinescu e Roman (*EJB Design Patterns*, Wiley, 2002).

Outras obras ainda tratam de campos de aplicação específicos. Entre elas estão a de Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004), Joshi (*C++ Design Patterns and Derivatives Pricing*, Cambridge University Press, 2004), Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002) e Schmidt e Rising (*Design Patterns in Communication Software*, Cambridge University Press, 2001).

Clássicos do arquiteto Christopher Alexander (*Notes on the Synthesis of Form*, Harvard University Press, 1964 e *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977) é uma leitura que vale a pena para um projetista de software que pretenda entender completamente os padrões de projeto.

Uma ampla gama de fontes de informação sobre projeto baseado em padrões se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevante para o projeto baseado em padrões, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

13

PROJETO DE WEBAPPS

CONCEITOS- -CHAVE

arquitetura	
de conteúdo ...	347
de objetos	345
arquitetura MVC.	349
arquitetura da	
WebApp	349
OOHDM	352
projeto	
arquitetural ...	346
de componentes .	352
de conteúdo ...	345
de metas	341
de navegação ..	350
estético	344
gráfico	344
pirâmide	342
qualidade	339

Em seu respeitado livro sobre projeto para Web, Jakob Nielsen [Nie00] afirma: “Existem, essencialmente, duas abordagens básicas para projeto: o ideal artístico de se expressar e o ideal de engenharia de resolver um problema para um cliente”. Durante a primeira década de desenvolvimento da Web, o ideal artístico foi a abordagem que muitos desenvolvedores escolheram. O projeto ocorreu em uma maneira *ad hoc* e foi usualmente conduzido à medida que o HTML era gerado. O projeto desenvolveu-se a partir de uma visão artística que evoluiu enquanto ocorria a construção de WebApps.

Mesmo hoje, muitos desenvolvedores para Web usam WebApps como ícone para indicar “projeto limitado”. Eles argumentam que o imediatismo e a volatilidade das WebApps aliviam o processo de projeto formal; que o projeto evolui à medida que uma aplicação é construída (codificada) e que relativamente pouco tempo deve ser gasto na criação de um modelo de projeto detalhado. Esse argumento tem seus méritos, mas apenas para WebApps relativamente simples. Quando conteúdo e função são complexos; quando o tamanho da WebApp engloba centenas ou milhares de objetos de conteúdo, funções e classes de análise; e quando o sucesso da WebApp terá um impacto direto no sucesso do negócio, o projeto não pode e não deve ser tratado de maneira superficial.

PANORAMA

O que é? O projeto para WebApps abrange atividades técnicas e não técnicas entre as quais: estabelecer a percepção e a aparência da WebApp, criar o layout estético da interface do usuário, definir a estrutura geral da arquitetura, desenvolver o conteúdo e a funcionalidade que residem na arquitetura e planejar a navegação que ocorre na WebApp.

Quem realiza? Engenheiros de aplicações para a Web, designers gráficos, desenvolvedores de conteúdo e outros interessados participam na criação de um modelo de projeto de uma WebApp.

Por que é importante? O projeto nos permite criar um modelo que pode ser avaliado em termos de qualidade e aperfeiçoado antes de serem gerados código e conteúdo; são realizados testes e usuários finais se envolvem em grande número. Projeto é onde se estabelece a qualidade de uma WebApp.

Quais são as etapas envolvidas? O projeto de WebApps abrange seis etapas principais orientadas por informações obtidas durante a modelagem de requisitos. O projeto de conteúdo usa o modelo de conteúdo (desenvolvido durante a análise) como base para estabelecer o projeto

de objetos de conteúdo. O projeto estético (também chamado design gráfico) estabelece o layout que o usuário final verá. O projeto da arquitetura se concentra na estrutura geral de hipermídia de todos os objetos de conteúdo e funções. O projeto da interface estabelece mecanismos de layout e interação que definem a interface do usuário. O projeto de navegação define como o usuário final navega pela estrutura de hipermídia, e o projeto de componentes representa a estrutura interna detalhada dos elementos funcionais da WebApp.

Qual é o artefato? Um modelo de projeto abrangendo questões de conteúdo, estética, arquitetura, interface, navegação e de projeto de componentes é o artefato primário gerado durante o projeto de uma WebApp.

Como garantir que o trabalho foi realizado corretamente? Cada elemento do modelo de projeto é revisado na tentativa de descobrir erros, inconsistências ou omissões. Além disso, são consideradas soluções alternativas, e o grau com que o modelo de projeto atual irá levar a uma implementação efetiva também é avaliado.

Essa realidade nos leva à segunda abordagem de Nielsen — “o ideal da engenharia de resolver um problema para um cliente”. A engenharia para Web¹ adota essa filosofia, e uma abordagem mais rigorosa para projeto de WebApps permite aos desenvolvedores alcançarem tal objetivo.

13.1 QUALIDADE DE PROJETO EM WEBAPPS

Projeto é a atividade da engenharia que conduz a um produto de alta qualidade. Isso nos leva a uma pergunta recorrente encontrada em todas as disciplinas de engenharia: O que é qualidade? Nesta seção examinaremos a resposta no contexto de desenvolvimento de WebApps.

Todo mundo que já navegou na Web ou usou uma Intranet corporativa tem opinião formada sobre o que faz uma WebApp ser “boa”. Os pontos de vista individuais variam muito. Alguns usuários adoram imagens chamativas; outros querem apenas texto. Alguns exigem informações detalhadas; outros desejam uma apresentação resumida. Alguns preferem ferramentas analíticas sofisticadas ou acesso a bancos de dados; outros preferem a simplicidade. Na realidade, a percepção do usuário de “excelência” (e a resultante aceitação ou rejeição da WebApp como consequência) talvez seja mais importante do que qualquer discussão técnica sobre a qualidade de WebApps.

Mas como a qualidade de uma WebApp é percebida? Quais atributos devem ser apresentados para atingir a excelência segundo a visão dos usuários finais e, ao mesmo tempo, apresentar as características técnicas de qualidade que lhe permitirá corrigir, adaptar, melhorar e suportar a aplicação no longo prazo?

Na realidade, todas as características técnicas da qualidade de projetos discutidas no Capítulo 8 e os atributos de qualidade gerais apresentados no Capítulo 14 se aplicam a WebApps. Entretanto, os atributos de qualidade gerais mais relevantes — usabilidade, funcionalidade, confiabilidade, eficiência e facilidade de manutenção — fornecem uma base útil para se avaliar a qualidade de sistemas baseados na Web.

Olsina e seus colegas [Ols99] prepararam uma “árvore de requisitos de qualidade” que identifica um conjunto de atributos técnicos — usabilidade, funcionalidade, confiabilidade, eficiência e facilidade de manutenção — que levam a WebApps de alta qualidade.² A Figura 13.1 sintetiza o trabalho desses pesquisadores. Os critérios citados na figura são de particular interesse caso você tenha de projetar, construir e manter WebApps no longo prazo.

Offutt [Off02] estende os cinco principais atributos de qualidade citados na Figura 13.1 acrescentando os seguintes atributos:

Segurança. As WebApps se tornaram altamente integradas a bancos de dados corporativos e governamentais críticos. Aplicações de comércio eletrônico extraem e depois armazenam informações confidenciais de clientes. Por essas e muitas outras razões, a segurança da WebApp é primordial em várias situações. A principal medida de segurança é a habilidade da WebApp e seu ambiente de servidor rechaçar acesso desautorizado e/ou frustrar um ataque mal-intencionado. Uma discussão detalhada sobre a segurança de WebApps está fora do escopo deste livro. Caso tenha maior interesse, veja [Vac06], [Kiz05] ou [Kal03].

Disponibilidade. Até mesmo a melhor WebApp não atenderá às necessidades dos usuários caso esteja indisponível. Em um sentido técnico, disponibilidade é a medida da porcentagem de tempo que uma WebApp está disponível para uso. O típico usuário final espera que as WebApps estejam disponíveis 24 horas por dia/7 dias por semana/365 dias por ano. Qualquer coisa abaixo

“Se os produtos são projetados para melhor atender às tendências naturais do comportamento humano, então as pessoas se sentirão mais satisfeitas, mais realizadas e mais produtivas.”

Susan
Weinschenk

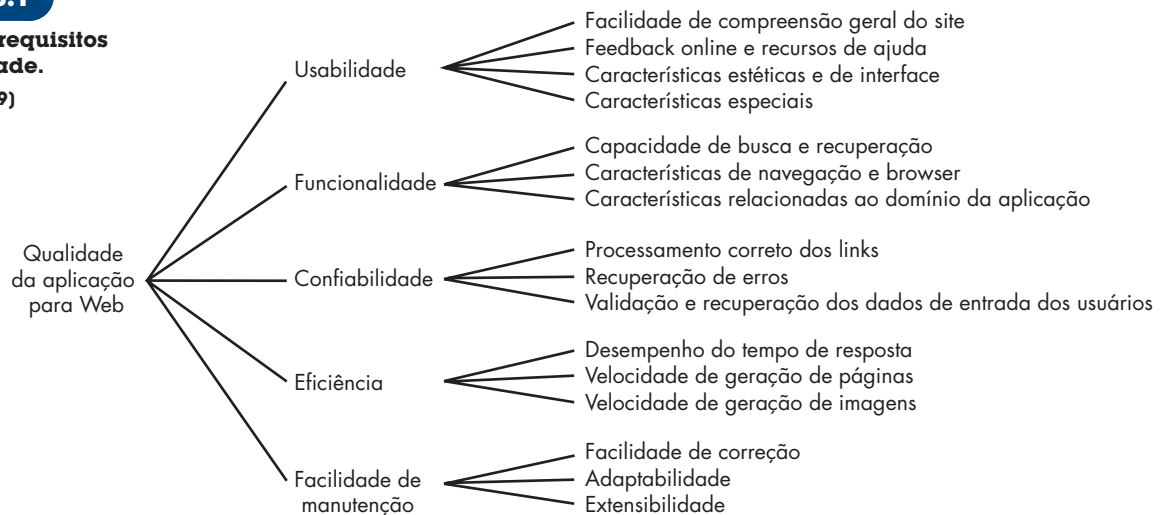
? Quais os
principais
atributos de
qualidade para as
WebApps?

¹ *Web engineering* [Pre08] é uma versão adaptada da abordagem de engenharia de software apresentada ao longo deste livro. Ela propõe uma estrutura ágil, embora disciplinada, para construção de sistemas e aplicações baseados na Web com alta qualidade.

² Esses atributos de qualidade são bem similares aos apresentados nos Capítulos 8 e 14. A implicação: características de qualidade são universais para todo software.

FIGURA 13.1**Árvore de requisitos de qualidade.**

Fonte: (Ols99)



disso é considerada inaceitável.³ Porém, *estar no ar* não é o único indicador de disponibilidade. Offutt [Off02] sugere que “as características de uso disponíveis em apenas um navegador ou uma plataforma” torna a WebApp indisponível para aqueles usam um navegador/plataforma diferente. O usuário invariavelmente irá procurar outra alternativa.

Escalabilidade. A WebApp e seus servidores podem ser escalados para atender 100, 1.000, 10.000 ou 100.000 usuários? A WebApp e os sistemas com os quais está interfaceando conseguem lidar com variação significativa de volume ou sua capacidade de resposta cairá significativamente (ou cessará de vez)? Não basta construir uma WebApp que seja bem-sucedida. É igualmente importante construir uma WebApp capaz de acomodar as responsabilidades inerentes ao sucesso (um número significativamente maior de usuários finais) e se tornar ainda mais bem-sucedida.

Tempo para colocação no mercado (*time-to-market*). Embora o tempo para colocação de um produto no mercado não seja um verdadeiro atributo de qualidade no sentido técnico, é uma medida de qualidade do ponto de vista comercial. A primeira WebApp a atender determina-se segmento de mercado em geral captura um número desproporcional de usuários finais.

**Projeto de WebApps — check-list de qualidade**

O check-list a seguir, adaptado das informações apresentadas em **Webreference.com**, fornece um conjunto de perguntas que ajudam tanto os projetistas de aplicações para Web como os usuários finais a avaliar a qualidade geral de uma WebApp:

- Opções de conteúdo e/ou função e/ou navegação podem ser ajustadas às preferências dos usuários?
- O conteúdo e/ou funcionalidade podem ser personalizados para a largura de banda em que o usuário se comunica?
- Imagens e outras mídias não textuais foram usadas adequadamente? Os tamanhos de arquivos gráficos são otimizados para eficiência de exibição?
- As tabelas são organizadas e dimensionadas para torná-las compreensíveis e exibidas de forma eficiente?
- O HTML é otimizado para eliminar ineficiências?
- O projeto geral de páginas é fácil de ler e navegar?
- Todos os links fornecem informações de interesse dos usuários?
- É provável que a maioria dos links tenha persistência na Web?
- A WebApp é equipada com recursos de administração de sites que incluem ferramentas para acompanhamento de uso, testes de links, busca de locais e segurança?

INFORMAÇÕES

3 Essa expectativa é, obviamente, irreal. As principais WebApps têm de programar *downtime* para correções e atualizações.

? O que devemos considerar ao avaliarmos a qualidade do conteúdo?

Bilhões de páginas Web se encontram disponíveis para aqueles em busca de informação. Mesmo as buscas na Web bem direcionadas resultam em uma avalanche de conteúdo. Com tantas fontes de informação para escolha, como o usuário pode avaliar a qualidade (por exemplo, veracidade, precisão, completude, oportunidade) do conteúdo apresentado em uma WebApp? Tillman [Til00] sugere um útil conjunto de critérios para avaliar a qualidade de conteúdos:

- O escopo e a profundidade do conteúdo podem ser facilmente determinados para garantir que atenda às necessidades dos usuários?
- A experiência e a autoridade dos autores do conteúdo podem ser facilmente identificadas?
- É possível determinar a atualidade do conteúdo, a última atualização e o que foi atualizado?
- O conteúdo e sua localização são estáveis (isto é, permanecerão na URL referida)?

Além dessas perguntas relacionadas a conteúdo, poderíamos adicionar o seguinte:

- O conteúdo é confiável?
- O conteúdo é exclusivo? A WebApp fornece algum benefício para aqueles que o usam?
- O conteúdo tem valor para a comunidade de usuários desejada?
- O conteúdo é bem organizado? Indexado? Facilmente acessível?

Os check-lists citados nesta seção representam apenas uma pequena amostra das questões que devem ser tratadas à medida que o projeto de uma WebApp evolui.

13.2 OBJETIVOS DE PROJETO

Em sua coluna regular sobre projeto para a Web, Jean Kaiser [Kai02] sugere um detalhado conjunto de objetivos de projeto que podem ser aplicados a praticamente qualquer WebApp independentemente do domínio de aplicação, do tamanho ou da complexidade:

“Só porque você pode não significa que deva.”

Jean Kaiser

“Para alguns, o projeto para Web enfoca o aspecto visual... Para outros, projeto para Web significa estruturar informações e navegação pelo espaço do documento. Outros até poderiam considerar que projeto para Web significa a tecnologia... Na realidade, o projeto inclui todas essas coisas e talvez até mais.”

Thomas Powell

Simplicidade. Embora possa parecer ultrapassado, o aforismo “tudo com moderação” se aplica às WebApps. Há uma tendência entre alguns projetistas de fornecer “em excesso” ao usuário final — conteúdo exaustivo, aspectos visuais excessivos, animação intrusiva, páginas Web enormes, a lista é longa. É melhor se esforçar pela moderação e simplicidade.

O conteúdo deve ser informativo, mas sucinto e deve usar um meio de entrega (por exemplo, texto, imagens, vídeo, áudio) apropriado para as informações que estão sendo entregues. A estética deve ser agradável, mas não opressiva (por exemplo, cores em demasia tendem a distrair o usuário em vez de melhorar a interação). A arquitetura deve atingir os objetivos da WebApp da maneira mais simples possível. A navegação deve ser simples e seus mecanismos intuitivamente óbvios para o usuário final. As funções devem ser fáceis de ser usadas e compreendidas.

Consistência. Esse objetivo de projeto se aplica a praticamente qualquer elemento do modelo de projeto. O conteúdo deve ser construído consistentemente (por exemplo, a formatação de texto e os estilos de fonte devem ser os mesmos ao longo de todos os documentos de texto; a arte gráfica deve ter aspecto, combinação de cores e estilo consistentes). O design gráfico (estética) deve apresentar um aspecto consistente ao longo de todas as partes da WebApp. O projeto da arquitetura deve estabelecer templates que levem a uma estrutura de hipermídia consistente. O projeto da interface deve definir modos de interação, navegação e exibição de conteúdo consistentes. Os mecanismos de navegação devem ser usados consistentemente por todos os elementos da WebApp. Como Kaiser [Kai02] observa: “Lembre-se de que para um visitante, um site é um lugar físico. Torna-se confuso se as páginas não forem consistentes no projeto”.

Identidade. A estética, a interface e o projeto de navegação de uma WebApp devem ser consistentes com o domínio de aplicação para o qual ela será construída. Um site para um grupo de *hip-hop* indubitavelmente terá uma percepção e aparência diferentes de uma WebApp desenhada para uma companhia financeira. A arquitetura da WebApp será completamente diferente, as

interfaces serão construídas para levar em conta diferentes categorias de usuários; a navegação será organizada para cumprir diferentes objetivos. Devemos (assim como outros colaboradores do projeto) trabalhar para estabelecer uma identidade para a WebApp por todo o projeto.

Robustez. Baseado na identidade estabelecida, em geral uma WebApp faz uma “promessa” implícita ao usuário, que espera funções e conteúdo robustos que são relevantes para as suas necessidades. Se esses elementos estiverem faltando ou insuficientes, é provável que a WebApp irá falhar.

Navegabilidade. Já citamos que a navegação deve ser simples e consistente. Ela também deve ser projetada de maneira intuitiva e previsível. Ou seja, o usuário deve entender como navegar pela WebApp sem ter de buscar links ou instruções para navegação. Por exemplo, se um campo de imagens ou ícones contiver ícones ou imagens selecionadas que serão usadas como mecanismos de navegação, estes devem ser identificados visualmente. Nada é mais frustrante que tentar encontrar o link ativo apropriado entre muitas imagens.

Também é necessário posicionar links para as funções e conteúdos mais importantes da WebApp em uma posição previsível em todas as páginas Web. Se for necessário rolar a página (e, normalmente, este é o caso), os links na parte superior e inferior da página tornam mais fácil as tarefas de navegação do usuário.

Apelo Visual. De todas as categorias de software, as aplicações para Web são, inquestionavelmente, as mais visuais, as mais dinâmicas e as mais estéticas. A beleza (apelo visual) é um conceito que varia segundo a ótica de quem a vê, porém, muitas características de projeto (por exemplo, a percepção e aspecto do conteúdo; o layout da interface; coordenação das cores; o equilíbrio entre texto, imagens e outras mídias; mecanismos de navegação) contribuem efetivamente para o apelo visual.

Compatibilidade. Uma WebApp será usada em uma variedade de ambientes (por exemplo, hardware diferente, tipos de conexão de Internet, sistemas operacionais, navegadores) e deve ser projetada para ser compatível com cada um deles.

13.3 UMA PIRÂMIDE DE PROJETO PARA WEBAPPS

“Se um site é perfeitamente utilizável mas falta um estilo de projeto elegante e adequado, ele falhará.”

Curt Cloninger

O que é projeto para WebApps? Essa simples questão é mais difícil de responder do que se pode imaginar. Em nosso livro [Pre08] sobre engenharia para Web, David Lowe e eu discutimos isso ao escrevermos:

A criação de um projeto eficaz exigirá, tipicamente, um conjunto de habilidades diversas. Às vezes, para pequenos projetos, um único desenvolvedor precisaria ter vários talentos. Para projetos maiores, seria aconselhável e/ou viável fazer uso da *expertise* de especialistas: engenheiros para aplicações para Web, designers gráficos, desenvolvedores de conteúdo, programadores, especialistas em bancos de dados, arquitetos da informação, engenheiros de rede, especialistas em segurança e aqueles que realizam testes. Fazer uso dessas diversas capacidades permite a criação de um modelo que pode ser avaliado em termos de qualidade e aperfeiçoado *antes* de o conteúdo e código serem gerados, os testes serem realizados e os usuários finais envolverem-se em grande número. Se análise é o momento em que se *estabelece a qualidade de uma WebApp*, então projeto é o momento em que a *qualidade é realmente incorporada*.

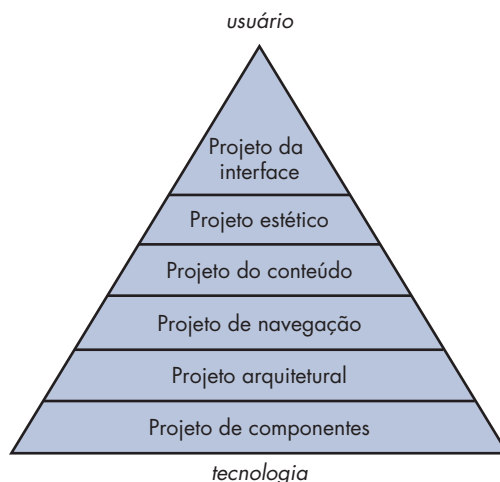
O mix apropriado de habilidades de projeto irá variar dependendo da natureza da WebApp. A Figura 13.2 apresenta uma pirâmide de projeto para WebApps. Cada nível da pirâmide representa uma ação de projeto descrita nas seções a seguir.

13.4 PROJETO DE INTERFACES PARA WEBAPPS

Quando um usuário interage com um sistema computacional, aplica-se um conjunto de princípios fundamentais e diretrizes de projeto primordiais. Estes foram discutidos no Capí-

FIGURA 13.2

Uma pirâmide de projeto para WebApps



tulo 11.⁴ Embora as WebApps apresentem alguns desafios especiais no projeto da interface do usuário, as diretrizes e princípios básicos se aplicam.

Um dos desafios no projeto da interface para WebApps é a natureza indeterminada do ponto de entrada do usuário. Ou seja, o usuário poderia entrar na WebApp pela localização “home” (por exemplo, a *homepage*) ou, por meio de um link, entrar em algum nível mais baixo da arquitetura da WebApp. Em alguns casos, a WebApp pode ser desenhada para redirecionar o usuário a uma localização home, mas se isso for indesejável, o projeto da WebApp deve fornecer recursos de navegação de interface que acompanhem todos os de conteúdo e estejam disponíveis independentemente de como o usuário entre no sistema.

Os objetivos de uma interface para WebApp são: (1) estabelecer uma janela consistente para o conteúdo e a funcionalidade fornecidos pela interface, (2) guiar o usuário através de uma série de interações com a WebApp e (3) organizar as opções de navegação e conteúdo disponíveis para o usuário. Para obtermos uma interface consistente, devemos primeiro usar a estética do projeto (Seção 13.5) para estabelecer um “aspecto” coerente. Isso abrange várias características, mas deve enfatizar o layout e a forma dos mecanismos de navegação. Para orientarmos a interação com o usuário, poderíamos fazer uso de uma metáfora⁵ apropriada que permita ao usuário ter um entendimento intuitivo da interface.

Para implementarmos opções de navegação, podemos selecioná-las de uma série de mecanismos de interação:

Quais mecanismos de interação estão disponíveis para os projetistas de WebApps?

- *Menus de navegação* — menus com palavras-chave (organizados vertical ou horizontalmente) que listam o conteúdo e/ou funcionalidade principais. Esses menus poderiam ser implementados para que o usuário possa escolher de uma hierarquia de subtópicos exibidos quando a opção de menu principal for escolhida.
- *Ícones* — botões, chaves e imagens similares que permitem ao usuário selecionar alguma propriedade ou especificar uma decisão.
- *Imagens* — alguma representação gráfica que é selecionável pelo usuário e implementa um link para um objeto de conteúdo ou funcionalidade da WebApp.

É importante notar que um ou mais desses mecanismos de controle devem ser fornecidos em todos os níveis da hierarquia de conteúdo.

⁴ A Seção 11.5 é dedicada ao projeto da interface de WebApps. Caso ainda não tenha feito, leia-a agora.

⁵ Nesse contexto, *metáfora* é uma representação (extraída da experiência real do usuário) que pode ser modelada no contexto da interface. Um exemplo simples poderia ser um controle deslizante usado para controlar o volume do áudio de um arquivo .mpg

13.5 PROJETO ESTÉTICO

Nem todo engenheiro de aplicações Web (ou engenheiro de software) tem talento artístico (estético). Caso se enquadre nessa categoria, contrate um designer gráfico experiente para realizar a tarefa de projeto estético.

“Constatamos que as pessoas avaliam rapidamente um site apenas pelo projeto visual.”

Stanford
Diretrizes para
Credibilidade na
Web



Os usuários tendem a tolerar rolagem vertical mais facilmente do que a horizontal. Evite formatos de página largos.

O projeto estético, também chamado *design gráfico*, é o esforço artístico que complementa os aspectos técnicos do projeto de WebApps. Sem ele, uma WebApp poderia ser funcional, mas não atraente. Com ele, uma WebApp atrai seus usuários para um mundo que os envolve em um nível físico, bem como intelectual.

Mas o que é estética? Há um velho ditado que diz: “A beleza existe segundo a ótica daquele que a vê”. Isso é particularmente apropriado quando se considera o projeto estético para WebApps. Para realizar um projeto estético efetivo, retorne à hierarquia de usuários desenvolvida como parte do modelo de requisitos (Capítulo 5) e pergunte: *Quem são os usuários da WebApp e que “visual” eles desejam?*

13.5.1 Problemas de layout

Toda página Web tem uma quantidade limitada de “terreno” que pode ser usado para dar suporte à estética não funcional, recursos de navegação, conteúdo de informação e funcionalidade dirigida ao usuário. O desenvolvimento desse terreno é planejado durante o projeto estético.

Assim como todas as questões estéticas, não há regras absolutas quando se desenvolve o layout da tela. Entretanto, vale a pena considerarmos uma série de diretrizes gerais para layout:

Não tenha medo de espaços em branco. É desaconselhável preencher cada centímetro de uma página Web com informação. O congestionamento visual resultante dificulta ao usuário identificar as informações ou os recursos necessários e cria um caos visual desagradável.

Enfatize o conteúdo. Afinal de contas, essa é razão para o usuário estar lá. Nielsen [Nie00] sugere que uma página Web típica deve ter 80% de conteúdo e o espaço restante dedicado à navegação e outros recursos.

Organize os elementos de layout de cima para baixo, da esquerda para a direita. A grande maioria dos usuários varrerá uma página Web de uma forma muito similar ao que faz ao ler a página de um livro — de cima para baixo, da esquerda para a direita.⁶ Se os elementos de layout tiverem prioridades específicas, os elementos de alta prioridade devem ser colocados na parte superior esquerda do espaço da página.

Agrupe a navegação, o conteúdo e as funções geograficamente dentro da página. Os seres humanos buscam padrões em quase tudo. Se não existirem padrões discerníveis em uma página Web, a frustração do usuário provavelmente aumentará (devido a buscas infrutíferas por informação necessária).

Não estenda seu espaço com a barra de rolagem. Embora muitas vezes a rolagem seja necessária, a maioria dos estudos indica que os usuários preferem não ficar rolando a página. É melhor reduzir o conteúdo da página Web ou apresentar o conteúdo necessário em várias páginas.

Considere a resolução e o tamanho da janela do navegador ao elaborar seu layout. Em vez de definir tamanhos fixos em um layout, o projeto deve especificar todos os itens de layout como uma porcentagem do espaço disponível [Nie00].

13.5.2 Questões de design gráfico

O design gráfico considera todos os aspectos visuais de uma WebApp. O processo de design gráfico começa com o layout (Seção 13.5.1) e prossegue com a consideração de combinações de cores gerais; tipos, tamanhos e estilos de texto; o uso de mídia complementar (por exemplo, áudio, vídeo, animação); e todos os demais elementos estéticos de uma aplicação.

Uma discussão completa sobre questões relativas ao design gráfico em WebApps está fora do escopo deste livro. Você pode obter dicas e diretrizes em muitos sites dedicados ao tema (por exemplo, www.graphic-design.com, www.grantasticdesigns.com, www.wpdfd.com) ou em um ou mais dos recursos impressos (por exemplo, [Roc06] e [Gor02]).

⁶ Existem exceções que se baseiam em questões culturais e do idioma usado, mas tal regra não se aplica à maioria dos usuários.

INFORMAÇÕES

**Sites bem projetados**

Algumas vezes, a melhor maneira de entender um bom projeto de uma WebApp é ver alguns exemplos. Em seu artigo, “The Top Twenty Web Design Tips”, Marcelle Toor (www.graphic-design.com/Web/feature/tips.html) sugere os seguintes sites como exemplos de design gráfico adequado:

www.creativepro.com/designresource/home/787.html — empresa de design dirigida por Primo Angeli

www.workbook.com — este site exibe trabalhos feitos por ilustradores e designers

www.pbs.org/riverofsong — série para TV pública e rádio sobre música americana

www.RKDINC.com — empresa de design com portfólio online e excelentes dicas sobre design

www.creativehotlist.com/index.html — excelente fonte de sites bem desenhados por agências de propaganda, empresas de artes gráficas e outros especialistas da comunicação

www.btdnyc.com — empresa de design dirigida por Beth Toudreau

13.6 PROJETO DE CONTEÚDO

“Bons designers são capazes de criar normalidade no caos; eles conseguem transmitir ideias claramente por meio da organização e manipulação de palavras e figuras.”

Jeffery Veen

O projeto de conteúdo aborda duas tarefas de projeto diferentes, cada uma delas tratadas por indivíduos com conjuntos de habilidades diferentes. Primeiro, são desenvolvidos uma representação de projeto para objetos de conteúdo e os mecanismos necessários para estabelecer seus relacionamentos. Além disso, são criadas as informações em um objeto de conteúdo específico. Esta última tarefa poderia ser realizada por redatores publicitários, designers gráficos e outros que geram o conteúdo a ser utilizado em uma WebApp.

13.6.1 Objetos de conteúdo

O relacionamento entre objetos de conteúdo definido como parte de um modelo de requisitos para a WebApp e os objetos de projeto representando o conteúdo é análogo ao relacionamento entre classes de análise e componentes de projeto descritos em capítulos anteriores. No contexto de projeto para WebApps, um objeto de conteúdo está mais alinhado com um objeto de dados para software tradicional. Um objeto de conteúdo possui atributos que incluem informações específicas de conteúdo (normalmente definidas durante a modelagem de requisitos da WebApp) e atributos de implementação exclusivos, especificados como parte do projeto.

Consideremos, por exemplo, uma classe de análise, **ComponenteDoProduto**, desenvolvida para o sistema de comércio eletrônico *CasaSegura*. O atributo de classe de análise, **descrição**, é representado com uma classe chamada **DescriçãoDeComponente** composta por cinco objetos de conteúdo, **DescriçãoDeMarketing**, **Fotografia**, **DescriçãoTécnica**, **Esquema** e **Vídeo**, indicados como objetos sombreados na Figura 13.3. As informações contidas no objeto de conteúdo são indicadas na forma de atributos. Por exemplo, **Fotografia** (uma imagem .jpg) possui os atributos **dimensão horizontal**, **dimensão vertical** e **estilo da borda**.

Associação e agregação⁷ UML podem ser usadas para representar relacionamentos entre os objetos de conteúdo. Por exemplo, a associação UML da Figura 13.3 indica que é usada uma classe **DescriçãoDeComponente** para cada instância da classe **ComponenteDoProduto**. **DescriçãoDeComponente** é composta pelos cinco objetos de conteúdo mostrados. Entretanto, a notação de multiplicidade indica que **Esquema** e **Vídeo** são opcionais (é possível que não haja nenhuma ocorrência), uma **DescriçãoDeMarketing** e uma **DescriçãoTécnica** são necessárias e usadas uma ou mais instâncias de **Fotografia**.

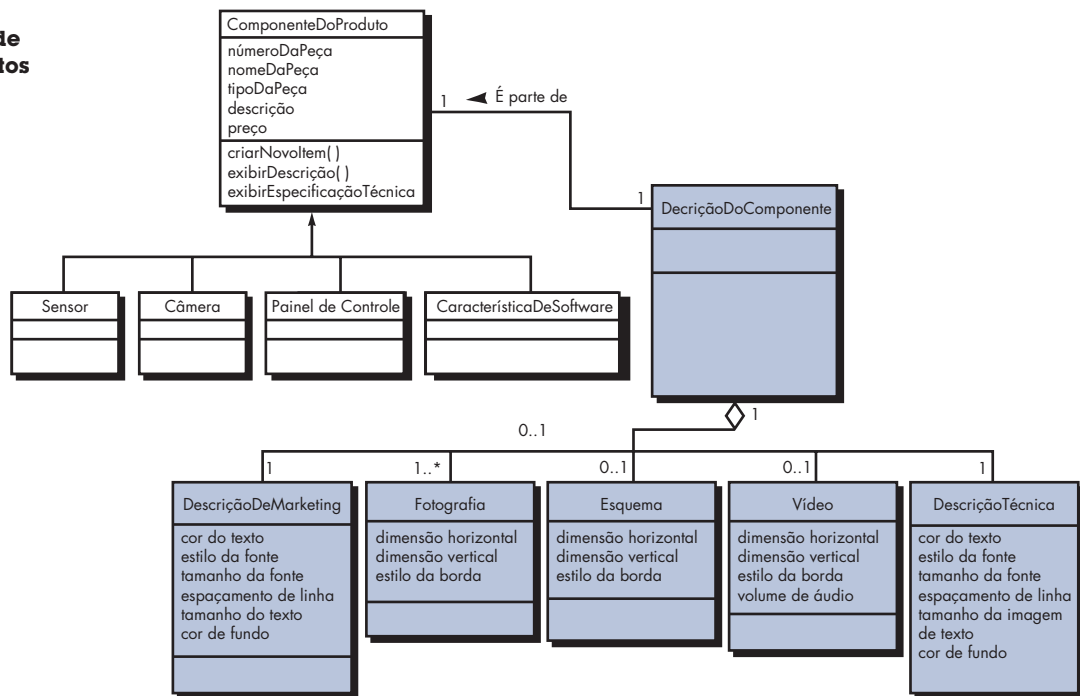
13.6.2 Questões de projeto de conteúdo

Assim que todos os objetos de conteúdo forem modelados, as informações que cada objeto deve fornecer devem passar por um processo de autoria e formatação para melhor atender às

⁷ Essas duas representações são discutidas no Apêndice 1.

FIGURA 13.3

Representação de projeto dos objetos de conteúdo



necessidades do cliente. A autoria de conteúdo é tarefa de especialistas que projetam o objeto de conteúdo fornecendo um resumo das informações a ser entregues e uma indicação dos tipos de objetos de conteúdo genéricos (por exemplo, texto descritivo, imagens, fotografias) usados para transmitir as informações. O projeto estético (Seção 13.5) também poderia ser aplicado para representar o aspecto visual para o conteúdo.

À medida que os objetos de conteúdo são projetados, eles são “agrupados” [Pow02] para formarem páginas Web. O número de objetos de conteúdo incorporados em uma única página é função das necessidades do usuário, das restrições impostas pela velocidade de *download* da conexão de Internet disponível e de restrições impostas pelo nível de rolagem que o usuário irá tolerar.

13.7 PROJETO ARQUITETURAL

“A estrutura arquitetural de um site bem desenhado nem sempre é aparente para o usuário — e nem deveria ser.”

Thomas Powell

O projeto arquitetural está ligado aos objetivos estabelecidos para uma WebApp, ao conteúdo a ser apresentado, aos usuários que visitarão a página e à filosofia de navegação que foi estabelecida. Como projetistas da arquitetura, temos de identificar a arquitetura de conteúdo e a arquitetura da WebApp. A *arquitetura de conteúdo*⁸ focaliza a maneira pela qual objetos de conteúdo (ou objetos compostos como páginas Web) são estruturados para apresentação e navegação. A *arquitetura de WebApps* lida com a maneira pela qual a aplicação é estruturada para administrar a interação com o usuário, tratar tarefas de processamento interno, navegação efetiva e apresentação de conteúdo.

Na maioria dos casos, o projeto arquitetural é conduzido em paralelo com os projetos da interface, estético e de conteúdo. Como a arquitetura da WebApp pode ter uma forte influência sobre a navegação, as decisões tomadas durante as etapas de projeto influenciarão o trabalho conduzido durante o projeto da navegação.

⁸ O termo *arquitetura da informação* também é usado para conotar estruturas que levam a uma melhor organização, atribuição de nomes, navegação e busca de objetos de conteúdo.

? Quais os tipos de arquitetura de conteúdo mais comuns?

13.7.1 Arquitetura de conteúdo

O projeto da arquitetura de conteúdo concentra-se na definição da estrutura geral de hiper-mídia da WebApp. Embora algumas vezes sejam criadas arquiteturas personalizadas, sempre temos a opção de escolher uma de quatro estruturas de conteúdo diferentes [Pow00]:

Estruturas lineares (Figura 13.4) são encontradas quando uma sequência de interações previsível (com certa variação ou desvios) é comum. Um exemplo clássico poderia ser uma apresentação de um tutorial em que as páginas de informação juntamente com as imagens, vídeos de curta duração ou áudio relacionados são apresentados apenas após as informações de pré-requisito terem sido apresentadas. A sequência de apresentação de conteúdo é predefinida e, em geral, linear. Outro exemplo poderia ser a sequência de preenchimento do pedido de um produto em que informações específicas devem ser especificadas em determinada ordem. Em tais casos, as estruturas da Figura 13.4 são apropriadas. À medida que o conteúdo e o processamento forem se tornando mais complexos, o fluxo puramente linear na parte esquerda da figura dá lugar a estruturas lineares mais sofisticadas, em que conteúdo alternativo poderia ser solicitado ou a ocorrência de uma mudança de direção para obter conteúdo complementar (a estrutura do lado direito da Figura 13.4).

Estruturas em grade (Figura 13.5) é uma opção de arquitetura que podemos aplicar quando conteúdo de WebApp pode ser organizado em categorias de duas (ou mais) dimensões. Consideremos, por exemplo, uma situação em que um site de comércio eletrônico vende tacos de golfe.

FIGURA 13.4

Estruturas lineares

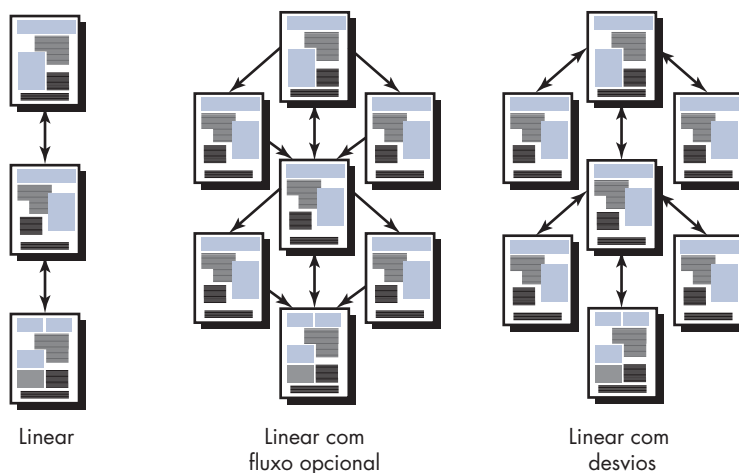
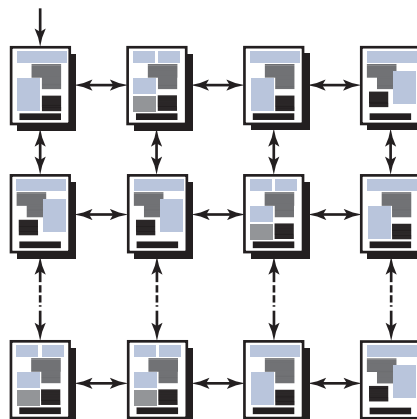


FIGURA 13.5

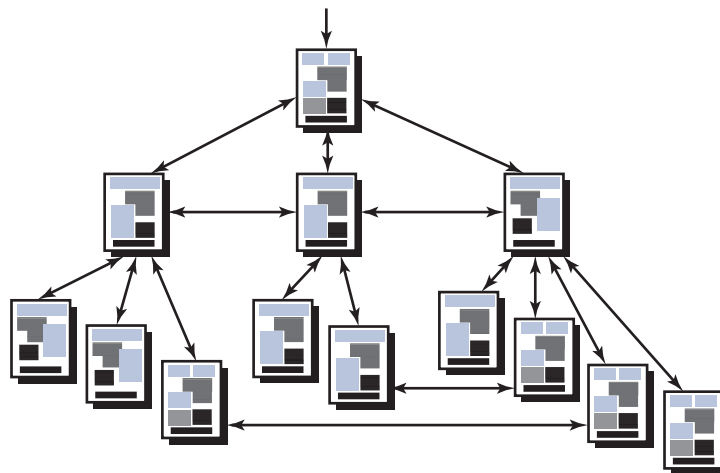
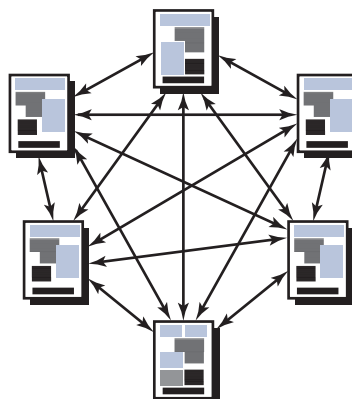
Estrutura em grade



A dimensão horizontal da grade representaria o tipo de taco a ser vendido (por exemplo, de madeira, ferro, embocador, curto). A dimensão vertical representa os produtos fornecidos pelos diversos fabricantes de tacos de golfe. Portanto, um usuário poderia navegar horizontalmente pela grade para encontrar a coluna tacos curtos e então verticalmente para examinar os produtos fornecidos por aqueles fabricantes que vendem tacos curtos. Essa arquitetura de WebApp é útil apenas quando é encontrado conteúdo altamente regular [Pow00].

As *estruturas hierárquicas* (Figura 13.6) são, sem dúvida nenhuma, a arquitetura para WebApp mais comum. Ao contrário das hierarquias de software particionadas discutidas no Capítulo 9, que encorajam o fluxo de controle apenas ao longo das ramificações verticais da hierarquia, uma estrutura hierárquica para WebApp pode ser projetada para possibilitar (via ramificação de hipertexto) o fluxo de controle horizontal ao longo de ramificações verticais da estrutura. Portanto, o conteúdo apresentado no ramo mais à esquerda da hierarquia pode ter links de hipertexto que levam diretamente a conteúdo existente no meio ou no ramo mais à direita da estrutura. Deve-se notar, entretanto, que embora tal ramificação possibilite rápida navegação pelo conteúdo de uma WebApp, ela pode gerar confusão para o usuário.

Uma *estrutura em rede ou “pura teia”* (Figura 13.7) é similar em muitos aspectos à arquitetura que evolui para sistemas orientados a objetos. Os componentes da arquitetura (nesse caso, páginas Web) são projetados de modo que possam passar o controle (via links de hipertexto) para praticamente qualquer outro componente do sistema. Essa abordagem possibilita uma flexibilidade de navegação considerável, mas, ao mesmo tempo, pode ser confusa para o usuário.

FIGURA 13.6**Estrutura hierárquica****FIGURA 13.7****Estrutura em rede**

As estruturas da arquitetura discutidas nos parágrafos anteriores podem ser combinadas para formar *estruturas compostas*. A arquitetura geral de uma WebApp pode ser hierárquica, porém, parte de sua estrutura poderia apresentar características lineares, enquanto outra poderia ser em rede. Nosso objetivo como projetista de arquiteturas é combinar a estrutura da WebApp com o conteúdo a ser apresentado e o processamento a ser realizado.

13.7.2 Arquitetura de uma WebApp

A arquitetura de uma WebApp descreve a infraestrutura que permite a uma aplicação ou sistema baseado na Web atingir seus objetivos de aplicação. Jacyntho e seus colegas [Jac02b] descrevem as características básicas dessa infraestrutura da seguinte maneira:

As aplicações devem ser construídas usando-se camadas em que diferentes preocupações são levadas em conta; em particular, os dados da aplicação devem ser separados do conteúdo da página (nós de navegação) e, por sua vez, os conteúdos devem estar claramente separados dos aspectos da interface (páginas).

Os autores sugerem uma arquitetura de projeto em três camadas que desassocia a interface da navegação e do comportamento da aplicação. Eles argumentam que manter a interface, a aplicação e a navegação separadas simplifica a implementação e aumenta a reutilização.

A arquitetura Modelo-Visão-Controlador (*Model-View-Controller MVC*) [Kra88]⁹ é uma de vários modelos de infraestrutura sugeridos para WebApps que desassocia a interface do usuário da funcionalidade e do conteúdo de informações de uma WebApp. O *modelo* (algumas vezes conhecido como “objeto-modelo”) contém todo o conteúdo e a lógica de processamento específicos à aplicação, inclusive todos os objetos de conteúdo, acesso a fontes de dados/informações externas e toda a funcionalidade de processamento específica para a aplicação. A *visão* contém todas as funções específicas à interface e possibilita a apresentação do conteúdo e lógica de processamento, inclusive todos os objetos de conteúdo, acesso a fontes de dados/informações externas e toda a funcionalidade de processamento requerida pelo usuário final. O *controlador* gerencia o acesso ao modelo e à visão e coordena o fluxo de dados entre eles. Em uma WebApp, “a visão é atualizada pelo controlador com dados do modelo baseados nas informações fornecidas pelos usuários” [WMT02]. Uma representação esquemática da arquitetura MVC está na Figura 13.8.

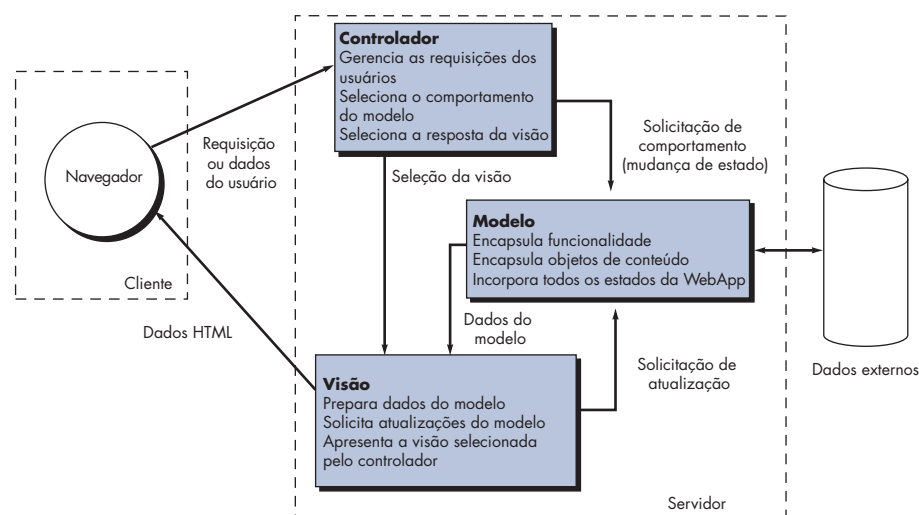
PONTO-CHAVE

A arquitetura MVC desacopla a interface do usuário da funcionalidade e do conteúdo de informação da WebApp.

FIGURA 13.8

A arquitetura MVC

Fonte: Adaptado de (Jac02)



⁹ Deve-se observar que a MVC é, na verdade, um padrão de projeto da arquitetura desenvolvido para o ambiente Smalltalk (veja www.cetus-links.org/oo_smalltalk.html) e que pode ser usado para qualquer aplicação interativa.

Referindo-se à figura, os dados ou solicitações de usuários são tratados pelo controlador, o qual também seleciona um objeto de visão aplicável baseado na solicitação do usuário. Uma vez determinado o tipo de solicitação, é transmitida uma solicitação de comportamento ao modelo, que implementa a funcionalidade ou recupera o conteúdo necessário para atender à solicitação. O objeto-modelo pode acessar dados armazenados em um banco de dados corporativo, como parte de um repositório de dados local ou de um conjunto de arquivos independentes. Os dados desenvolvidos pelo modelo devem ser formatados e organizados pelo objeto de visão apropriado e transmitidos do servidor de aplicações de volta para o navegador instalado no cliente para exibição na máquina do usuário.

Em muitos casos, a arquitetura da WebApp é definida no contexto do ambiente de desenvolvimento em que a aplicação será implementada. Caso tenha maior interesse, veja [Fow03] para uma discussão dos ambientes de desenvolvimento e seus papéis no projeto de arquiteturas para aplicações para Web.

13.8 PROJETO DA NAVEGAÇÃO

“Apenas espere, Maria, até que a lua surja e então iremos ver as migalhas de pão que espalhei pelo chão; elas nos indicarão o caminho de volta para casa.”

João e Maria

Assim que a arquitetura da WebApp tiver sido estabelecida e os componentes (páginas, scripts, applets e outras funções de processamento) da arquitetura identificados, temos de definir os percursos de navegação que permitirão aos usuários acessarem o conteúdo e as funções da WebApp. Para tanto, devemos: (1) identificar a semântica de navegação para diferentes usuários do site e (2) definir a mecânica (sintaxe) para atingir a navegação.

13.8.1 Semântica de navegação

Assim como muitas atividades de projeto para WebApps, o projeto da navegação começa considerando-se a hierarquia dos usuários e casos de uso relativos (Capítulo 5) desenvolvidos para cada categoria de usuário (ator). Cada ator deve usar a WebApp de forma ligeiramente distinta e, portanto, apresentar diferentes necessidades de navegação. Além disso, os casos de uso desenvolvidos para cada ator irão definir um conjunto de classes que engloba um ou mais objetos de conteúdo ou funções de WebApp. À medida que cada usuário interage com a WebApp, ele encontra uma série de *unidades semânticas de navegação* (*navigation semantic units, NSUs*) — “um conjunto de informações e estruturas de navegação relacionadas que colaboram no cumprimento de um subconjunto de requisitos de usuário relacionados” [Cac02].

Uma NSU é composta por um conjunto de elementos de navegação denominado *modos de navegação* (*ways of navigating, WoN*) [Gna99]. Um WoN representa o melhor percurso de navegação para atingir uma meta de navegação para um tipo de usuário específico. Cada WoN é organizado como um conjunto de *nós de navegação* (*navigational nodes, NN*) interligados por links de navegação. Em alguns casos, um link de navegação poderia ser uma outra NSU. Consequentemente, a estrutura geral de navegação para uma WebApp poderia ser organizada como uma hierarquia de NSUs.

Para ilustrarmos o desenvolvimento de uma NSU, consideremos o caso de uso **Selecionar Componentes do CasaSegura**:

Caso de uso: Selecionar Componentes do CasaSegura

A WebApp irá recomendar componentes de produto (por exemplo, painéis de controle, sensores, câmeras) e outras características (por exemplo, funcionalidade baseada em PC implementada por software) para cada cômodo e para a entrada externa. Se eu solicitar alternativas, a WebApp irá fornecê-las, caso elas existam. Serei capaz de obter informações descritivas e de preços para cada componente do produto. A WebApp criará e exibirá uma lista de materiais à medida que for selecionando vários componentes. Serei capaz de dar um nome à lista de materiais e salvá-la para referência futura (veja o caso de uso **Salvar Configuração**).

Os itens sublinhados na descrição do caso de uso representam classes e objetos de conteúdo que serão incorporados em uma ou mais NSUs que possibilitarão a um novo cliente representar o cenário descrito no caso de uso **Selecionar Componentes do CasaSegura**.

PONTO-CHAVE

Uma NSU descreve os requisitos de navegação para cada caso de uso. Em essência, a NSU mostra como um ator se movimenta pelos objetos de conteúdo ou funções da WebApp.

“O problema da navegação em um site é conceitual, técnico, espacial, filosófico e logístico. Consequentemente, as soluções tendem a exigir combinações improvisadas e complexas de arte, ciências e psicologia organizacional.”

Tim Horgan

A Figura 13.9 representa uma análise semântica parcial da navegação implícita no caso de uso **Selecionar Componentes do CasaSegura**. Usando a terminologia introduzida anteriormente, a figura também representa uma forma de navegação (WoN) para a WebApp **CasaSeguraGarantida.com**. São mostradas importantes classes de domínio do problema com objetos de conteúdo selecionados (nesse caso, o pacote de objetos de conteúdo chamado **DescriçãoDeComponente**, um atributo da classe **ComponenteDeProduto**). Esses itens são nós de navegação. Cada uma das setas representa um link de navegação¹⁰ e é rotulado com a ação iniciada pelo usuário que faz com que o link ocorra.

Podemos criar uma NSU para cada caso de uso associado ao papel de cada usuário. Por exemplo, um **novo cliente** do **CasaSeguraGarantida.com** poderia ter casos de uso diferentes, todos resultando no acesso a diferentes informações e funções de WebApp. É criada uma NSU para cada objetivo.

Durante os estágios iniciais do projeto de navegação, a arquitetura de conteúdo da WebApp é avaliada para determinar um ou mais WoNs para caso de uso. Conforme citado, um WoN identifica nós de navegação (por exemplo, conteúdo) e links que possibilitam a navegação entre eles. Os WoNs são organizados em NSUs.

13.8.2 Sintaxe de navegação

À medida que o projeto prossegue, sua tarefa é definir a mecânica de navegação. Estão disponíveis várias abordagens para implementação de cada NSU:

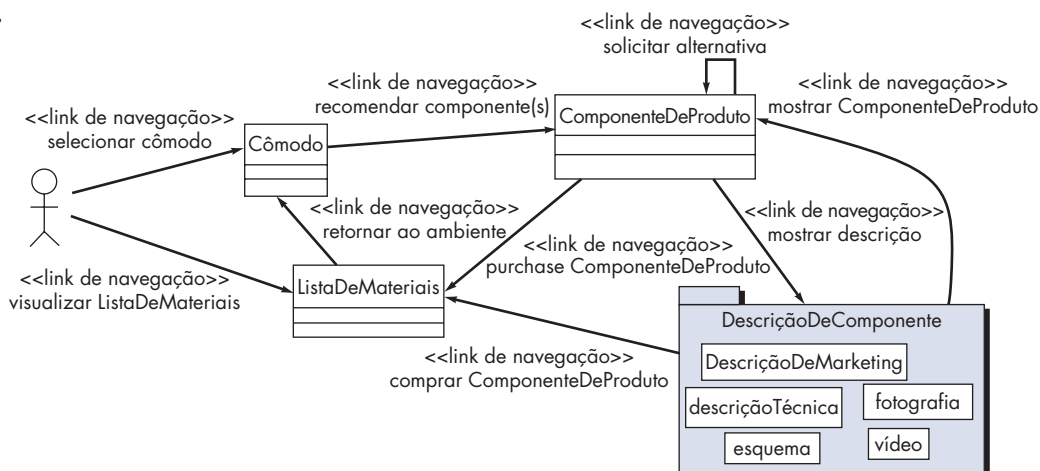
- *Link de navegação individual* — inclui links de texto, ícones, botões e chaves e metáforas gráficas. Temos de escolher links de navegação apropriados para o conteúdo e consistentes com a heurística que levam a uma interface de projeto de alta qualidade.
- *Barra de navegação horizontal* — lista as principais categorias funcionais ou de conteúdo em uma barra contendo links apropriados. Em geral, são listadas entre quatro e sete categorias.
- *Coluna de navegação vertical* — (1) lista as principais categorias funcionais ou de conteúdo ou (2) lista praticamente todos os principais objetos de conteúdo de uma WebApp. Caso escolha a segunda opção, tais colunas de navegação podem “expandir-se” para apresentar objetos de conteúdo como parte da hierarquia (selecionar uma entrada na coluna original fará com que uma expansão liste uma segunda camada de objetos de conteúdo relacionados).



Na maioria das situações, opte por mecanismos de navegação horizontais ou verticais, mas não ambos.

FIGURA 13.9

Criação de uma NSU



¹⁰ Esses são, algumas vezes, conhecidos como *links de semântica de navegação* (*navigational semantic links, NSL*) [Cac02].



O mapa de um site deve ser acessível de qualquer página. O próprio mapa deve ser organizado de modo que a estrutura de informações da WebApp estejam prontamente visíveis.

- *Guias* — uma metáfora que nada mais é que uma variação de uma barra ou coluna de navegação, representando categorias funcionais ou de conteúdo, como páginas de guias selecionadas quando um link for necessário.
- *Mapas de sites* — fornecem um sumário completo para navegação a todos os objetos de conteúdo e funcionalidade contidos na WebApp.

Além de escolhermos a mecânica de navegação, também podemos estabelecer convenções e ferramentas de ajuda de navegação adequadas. Por exemplo, ícones e links gráficos devem ter um aspecto “clicável” por meio de elevação das arestas para conferir à imagem um aspecto tridimensional. Deve-se implementar feedback sonoro ou visual para dar ao usuário uma indicação de que a opção de navegação foi escolhida. Para navegação baseada em elementos textuais, devem-se usar cores para indicar links de navegação e fornecer uma indicação dos links já navegados. Essas são apenas algumas das dezenas de convenções de projeto que tornam a navegação mais amigável.

13.9 PROJETO DOS COMPONENTES

As WebApps modernas oferecem funções de processamento cada vez mais sofisticadas que: (1) executam processamento localizado para gerar recursos de navegação e conteúdo de forma dinâmica, (2) fornecem recursos de cálculo ou processamento de dados apropriados para o campo de aplicação da WebApp, (3) fornecem sofisticadas consultas e acesso a bancos de dados e (4) estabelecem interfaces de dados com sistemas corporativos externos. Para alcançarmos essas (e muitas outras) capacidades, temos de projetar e construir componentes de programa que sejam idênticos em sua forma aos componentes para software tradicional.

Os métodos de projeto discutidos no Capítulo 10 se aplicam aos componentes para WebApp com poucas, se realmente alguma, modificações. O ambiente de implementação, as linguagens de programação e os padrões de projeto, estruturas e software podem variar um pouco, porém, a abordagem de projeto geral permanece a mesma.

13.10 MÉTODO DE PROJETO DE HIPERMÍDIA ORIENTADO A OBJETOS (OBJECT-ORIENTED HYPERMEDIA DESIGN METHOD – OOHDM)

Foram propostos vários métodos de projeto para aplicações para Web ao longo da última década. Até hoje, nenhum método exclusivo atingiu uma predominância.¹¹ Nesta seção apresentamos uma visão breve de um dos métodos de projeto para WebApps mais amplamente discutidos — o OOHDM.

Daniel Schwabe e seus colegas [Sch95, Sch98b] propuseram originalmente o *método para projeto de hipermídia orientado a objetos* (Object-Oriented Hypermedia Design Method, OOHDM), que é composto por quatro atividades de projeto diversas: atividade conceitual, projeto da navegação, projeto da interface abstrata e implementação. Uma síntese dessas atividades de projeto é mostrada na Figura 13.10 e discutida brevemente nas seções a seguir.

13.10.1 Projeto conceitual para o OOHDM

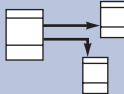
O *projeto conceitual* para o OOHDM cria uma representação dos subsistemas, classes e relações que definem o domínio de aplicação para a WebApp. Poderíamos usar a UML¹² para criar diagramas de classes apropriados, agregações e representações de classes compostas, diagramas de colaboração e outras informações que descrevem o domínio de aplicação.

¹¹ Na realidade, relativamente poucos desenvolvedores de aplicações para Web usam um método específico ao projetar uma WebApp. Espera-se que essa abordagem de projeto *ad hoc* mude com o passar do tempo.

¹² O OOHDM não prescreve uma notação específica; entretanto, o uso da UML é comum quando se aplica esse método.

FIGURA 13.10**Resumo do método OOHDM**

Fonte: Adaptado de (Sch95)

	 Projeto conceitual	 Projeto da navegação	 Projeto da interface abstrata	 Implementação
Artefatos	Classes, subsistemas, relacionamentos, atributos	Links de nós, estruturas de acesso, contextos de navegação, transformações de navegação	Objetos de interface abstrata, respostas a eventos externos, transformações	WebApp executável
Mecanismos de projeto	Classificação, composição, agregação, generalização, especialização	Mapeamento entre objetos conceituais e de navegação	Mapeamento entre objetos de navegação e perceptíveis	Recurso fornecido pelo Ambiente-alvo
Interesses de projeto	Modelagem semântica do domínio de aplicação	Leva em conta o perfil e as tarefas do usuário. Ênfase em aspectos cognitivos.	Modelagem de objetos perceptíveis, implementando as metáforas escolhidas. Descreve a interface para os objetos de navegação.	Correção; desempenho da aplicação; completude

Como um exemplo simples do projeto conceitual para o OOHDM, consideremos a aplicação de comércio eletrônico **CasaSeguraGarantida.com**. Um “esquema conceitual” parcial está na Figura 13.11. Os diagramas de classes, as agregações e informações relativas desenvolvidas como parte da análise de WebApps são reutilizados durante o projeto conceitual para representar relações entre as classes.

13.10.2 Projeto da navegação para o OOHDM

O *projeto da navegação* identifica um conjunto de “objetos” obtidos das classes definidas no projeto conceitual. São definidas “classes de navegação” ou “nós” para encapsular esses objetos. Poderíamos usar a UML para criar casos de uso, tabelas de estados e diagramas de sequências apropriados — todas as representações que nos ajudam a compreender melhor os requisitos de navegação. Além desses, podem ser usados os padrões de projeto para o projeto de navegação à medida que o projeto é desenvolvido. O OOHDM usa um conjunto de classes de navegação predefinido — nós, links, âncoras e estruturas de acesso [Sch98b]. As estruturas de acesso são mais elaboradas e incluem mecanismos como um índice de WebApps, um mapa do site ou um *tour* orientado.

Assim que as classes de navegação forem definidas, o OOHDM “estrutura um espaço de navegação por meio do agrupamento de objetos de navegação em conjuntos denominados contextos” [Sch98b]. Um *contexto* inclui uma descrição da estrutura de navegação local, restrição imposta ao acesso de objetos de conteúdo e métodos (operações) necessários para efetuar o acesso a objetos de conteúdo. É desenvolvido um template de contextos (análogo aos cartões CRC discutidos no Capítulo 6) que pode ser usado para acompanhar os requisitos de navegação de cada categoria de usuário através dos vários contextos definidos no OOHDM. Desse modo, vêm à tona caminhos de navegação específicos (aquilo que denominamos WoN na Seção 13.8.1).

13.10.3 Projeto da interface abstrata e implementação

A ação de *projeto da interface abstrata* especifica os objetos de interface que o usuário vê à medida que ocorre a interação da WebApp. Um modelo formal de objetos de interface, denominado *visão abstrata de dados* (*abstract data view, ADV*), é usado para representar o relacio-

o uso de elementos gráficos e decisões estéticas relacionadas. Um conjunto de diretrizes para design gráfico fornece a base para uma abordagem ao projeto.

O projeto de conteúdo define o layout, a estrutura e um resumo para todo o conteúdo apresentado como parte da WebApp e estabelece as relações entre os objetos de conteúdo. O projeto de conteúdo começa com a representação dos objetos de conteúdo, suas associações e relações. Um conjunto de rudimentos de navegação estabelece a base para o projeto de navegação.

O projeto da arquitetura identifica a estrutura de hipermídia geral para a WebApp e engloba tanto a arquitetura de conteúdo quanto a arquitetura da WebApp. Os estilos da arquitetura para conteúdo incluem estruturas lineares, em grade, hierárquicas e em rede. A arquitetura da WebApp descreve uma infraestrutura que permite a um sistema ou aplicação baseados na Web atingir os objetivos de seu domínio de aplicação.

Projeto de navegação representa um fluxo de navegação entre objetos de conteúdo e para todas as funções da WebApp. A semântica da navegação é definida descrevendo-se um conjunto de unidades semânticas de navegação. Cada unidade é composta por formas de navegação e links e nós de navegação. A sintaxe de navegação representa os mecanismos usados para efetuar a navegação descrita como parte da semântica.

O projeto de componentes desenvolve a lógica de processamento detalhada para implementar os componentes funcionais que implementam uma função de WebApp completa. As técnicas de projeto descritas no Capítulo 10 se aplicam à criação de componentes para WebApps.

O método para projeto de hipermídia orientado a objetos (OOHDM) é um dos vários métodos propostos para projeto de WebApps. O OOHDM sugere um processo de projeto que abrange os projetos conceitual, de navegação e da interface abstrata, bem como a implementação.

PROBLEMAS E PONTOS A PONDERAR

13.1. Por que o “ideal artístico” é uma filosofia de projeto insuficiente ao se construírem WebApps modernas? Existe algum caso em que o ideal artístico é a filosofia a ser seguida?

13.2. Neste capítulo escolhemos uma ampla gama de atributos de qualidade para WebApps. Escolha os três que você acredita serem os mais importantes e defenda um argumento que explique por que cada um deve ser enfatizado no trabalho de projeto para WebApps.

13.3. Acrescente pelo menos cinco outras questões ao projeto de WebApps check-list de qualidade apresentado na Seção 13.1.

13.4. Você é um projetista de WebApps da *FutureLearning Corporation*, uma empresa de aprendizagem à distância. Você pretende implementar um “mecanismo de aprendizagem” baseado na Internet que deixará à disposição conteúdo de cursos para os estudantes. O mecanismo de aprendizagem fornece a infraestrutura básica para transmissão de conteúdo de aprendizagem sobre qualquer tema (os projetistas de conteúdo prepararão o conteúdo apropriado). Desenvolva um protótipo de projeto da interface para o mecanismo de aprendizagem.

13.5. Qual o site mais esteticamente atraente que você já visitou até hoje e por quê?

13.6. Considere o objeto de conteúdo **Pedido**, gerado assim que o usuário do **CasaSeguraGarantida.com** tenha completado a escolha de todos os componentes e esteja pronto para finalizar sua compra. Desenvolva uma descrição UML para **Pedido** com todas as representações de projeto apropriadas.

13.7. Qual a diferença entre arquitetura de conteúdo e arquitetura de uma WebApp?

13.8. Reconsiderando o “mecanismo de aprendizagem” da *FutureLearning* descrito no Problema 13.4, escolha uma arquitetura de conteúdo que seja apropriada para a WebApp. Discuta a razão para ter feito tal escolha.

13.9. Use UML para desenvolver três ou quatro representações de projeto para objetos de conteúdo que seriam encontrados enquanto o “mecanismo de aprendizagem” descrito no Problema 13.4 é desenhado.

13.10. Pesquise mais sobre a arquitetura MVC e decida se seria ou não a arquitetura de Web App apropriada para o “mecanismo de aprendizagem” discutido no Problema 13.4.

13.11. Qual a diferença entre sintaxe de navegação e navegação semântica?

13.12. Defina duas ou três NSUs para a WebApp **CasaSeguraGarantida.com**. Descreva cada uma delas com certo nível de detalhe.

13.13. Redija um breve artigo sobre um método de projeto de hipermídias que não seja o OOHDM.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Van Duyne e seus colegas (*The design of Sites*, 2. ed., Prentice Hall, 2007) escreveram um livro completo que cobre os mais importantes aspectos do processo de projeto de WebApps. Modelos de processo de projeto e padrões de projeto são vistos em detalhe. Wodtke (*Information Architecture*, New Riders Publishing, 2003), Rosenfeld e Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002) e Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) tratam arquitetura de conteúdo e outros tópicos.

Embora tenham sido escritos centenas de livros sobre “Web design”, poucos discutem quaisquer métodos técnicos significativos para a realização do trabalho de projeto. Na melhor das hipóteses, são apresentadas várias diretrizes úteis para projeto de WebApps, são mostrados exemplos de páginas Web e programação Java que valem a pena, e são discutidos detalhes técnicos importantes para a implementação de WebApps modernas. Entre as várias ofertas dessa categoria temos livros como os de Sklar (*Principles of Web Design*, 4. ed., Course Technology, 2008), McIntire (*Visual Design for the Modern Web*, New Riders Press, 2007), Niederst (*Web Design in a Nutshell*, 3. ed., O'Reilly, 2006), Eccher (*Advanced Professional Web Design*, Charles River Media, 2006), Cederholm (*Bulletproof Web Design*, New Riders Press, 2005) e Shelly e seus colegas (*Web Design*, 2. ed., Course Technology, 2005). A discussão enciclopédica de Powell [Pow02] bem como a aprofundada discussão de Nielsen [Nie00] sobre projeto também são livros que valem a pena fazer parte de qualquer biblioteca.

Livros como os de Beaird (*The Principles of Beautiful Web Design*, SitePoint, 2007), Clarke e Holzschlag (*Transcending CSS: The Fine Art of Web Design*, New Riders Press, 2006) e Golbeck (*Art Theory for Web Design*, Addison Wesley, 2005) enfatizam o projeto estético e devem ser lidos por profissionais com pouca experiência no assunto.

A visão ágil de projeto (e outros tópicos) para WebApps é apresentada por Wallace e seus colegas (*Extreme Programming for Web Projects*, Addison-Wesley, 2003). Conallen (*Building Web Applications with UML*, 2. ed., Addison-Wesley, 2002) e Rosenberg e Scott (*Applying Use-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) apresentam exemplos detalhados de WebApps modeladas com o emprego da UML.

Também são mencionadas técnicas de projeto no contexto de livros escritos para ambientes de desenvolvimento específicos. Os leitores interessados devem examinar obras sobre HTML, CSS, J2EE, Java, .NET, XML, Perl, Ruby on Rails, Ajax e uma série de aplicações para criação de WebApps (*Dreamweaver*, *HomePage*, *Frontpage*, *GoLive*, *Macromedia Flash* etc.) para obter úteis informações sobre design.

Uma ampla gama de fontes de informação sobre projeto de WebApps se encontra à disposição na Internet. Uma lista atualizada de referências na Web, relevante para o projeto de WebApps, pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

GESTÃO DA QUALIDADE

Nesta parte do livro serão vistos os princípios, as técnicas e os conceitos aplicados ao gerenciamento e controle da qualidade de software. As seguintes questões são tratadas nos próximos capítulos:

- Quais são as características genéricas de um software de alta qualidade?
- Como revisar a qualidade e como são realizadas revisões eficazes?
- O que é garantia da qualidade de software?
- Quais são as estratégias aplicáveis aos testes de software?
- Quais métodos são usados para projetar casos de teste eficazes?
- Existem métodos práticos para garantir que um software está correto?
- Como gerenciar e controlar mudanças que sempre ocorrem quando um software é criado?
- Que medidas e métricas podem ser usadas para avaliar a qualidade dos modelos de requisitos e de projeto, código-fonte e casos de teste?

Assim que todas essas questões forem respondidas, garante-se uma melhor preparação para a produção de software de alta qualidade.

**CONCEITOS-
-CHAVE**

ações	
administrativas...	369
custo da	
qualidade	366
dimensões de	
qualidade	360
fatores de	
qualidade	361
o dilema da	
qualidade	365
qualidade	359
responsabilidade	
civil	368
riscos	368
segurança	368
software bom	
o suficiente	365
visão quantitativa	364

O clamor por maior qualidade de software começou realmente quando o software passou a se tornar cada vez mais integrado em todas as atividades de nossas vidas. Na década de 1990, as principais empresas reconheciam que bilhões de dólares por ano estavam sendo desperdiçados em software que não apresentava as características e as funcionalidades prometidas. Pior ainda, tanto o governo como as empresas ficavam cada vez mais preocupados com o fato de que uma falha grave de software poderia inutilizar importantes infraestruturas, aumentando o custo em dezenas de bilhões. Na virada do século, a *CIO Magazine* [Lev01] anunciou em manchete: “Chega de desperdiçar US\$ 78 bilhões por ano”, lamentando o fato de que “as empresas americanas gastavam bilhões em softwares que não faziam o que supostamente deveriam fazer”. A *InformationWeek* [Ric01] anunciou a mesma preocupação:

Apesar das boas intenções, código mal feito continua a ser o “fantasma” do mercado de software, sendo responsável por até 45% do tempo de inatividade dos sistemas computacionais e custando às empresas americanas cerca de US\$ 100 bilhões no último ano em termos de manutenção e redução da produtividade, afirma o Standish Group, uma empresa de pesquisa de mercado. Isso não inclui o custo da perda de clientes insatisfeitos. Pelo fato de as empresas de TI escreverem aplicações que dependem de pacotes de software de infraestrutura, código de má qualidade pode causar estragos em aplicações personalizadas bem como...

Qual o prejuízo causado por software de má qualidade? As definições variam, mas especialistas dizem que bastam três ou quatro defeitos a cada 1.000 linhas de código para fazer com que um programa execute de forma inadequada. Acrescente a isso que a maioria dos programadores insere cerca de um erro a cada 10 linhas de código escrito, multiplicados por milhões de linhas de código em vários produtos comerciais. Assim, deduz-se que o custo dos fornecedores de software será de pelo menos a metade dos seus orçamentos para a realização dos testes e correção dos erros. Percebeu o tamanho do problema?

PANORAMA

O que é? A resposta não é tão simples quanto se imagina. Sabe-se o que é qualidade ao vê-la e, mesmo assim, pode ser algo difícil de definir. Porém, para software de computador, qualidade é algo que tem de ser definido e é isso o que é feito neste capítulo.

Quem realiza? Todo mundo: os engenheiros de software, gerentes, todos os interessados; todos os envolvidos na gestão de qualidade são responsáveis por ela.

Por que é importante? Pode-se fazer certo da primeira vez ou então fazer tudo de novo. Se uma equipe de software enfatizar a qualidade em todas as atividades de engenharia de software, ela reduzirá a quantidade de reformulações que terá de fazer. Isso resulta em custos menores e, mais importante ainda, menor tempo para a colocação do produto no mercado.

Quais são as etapas envolvidas? Para obter software de alta qualidade, devem ocorrer quatro atividades: processo e prática comprovados de engenharia de software, gerenciamento consistente de projetos, controle global de qualidade e a presença de uma infraestrutura para garantir a qualidade.

Qual o artefato? Software que atenda às necessidades do cliente, execute de forma precisa e confiável e gere valor para todos aqueles que o utilizam.

Como garantir que o trabalho foi realizado corretamente? Acompanhando a qualidade por meio da verificação dos resultados de todas as atividades de controle de qualidade, medindo a qualidade efetuando a verificação de erros antes da entrega e de defeitos que acabaram escapando e indo para a produção.

Em 2005, a *ComputerWorld* [Hil05] lamentou que “software de má qualidade está em praticamente todas as organizações que usam computadores, provocando horas de trabalho perdidas durante o tempo em que a máquina fica parada, dados perdidos ou corrompidos, oportunidades de vendas perdidas, custos de suporte e manutenção de TI elevados e baixa satisfação do cliente”. Um ano mais tarde, a *InfoWorld* [Fos06] escreveu sobre “o estado de penúria da qualidade de software”, relatando que o problema da qualidade não havia melhorado.

Hoje em dia, a qualidade de software continua a ser um problema, mas a quem culpar? Os clientes culpam os desenvolvedores, argumentando que práticas descuidadas levam a um software de baixa qualidade. Os desenvolvedores de software culpam os clientes (e outros interessados), argumentando que datas de entrega absurdas e um fluxo contínuo de mudanças os forçam a entregar software antes de eles estarem completamente validados. Quem está com a razão? *Ambos* — e esse é o problema. Neste capítulo, considera-se a qualidade de software um conceito e examina-se por que vale a pena considerá-la seriamente toda vez que as práticas de engenharia de software forem aplicadas.

14.1 O QUE É QUALIDADE?

Em seu livro místico, *Zen and the Art of Motorcycle Maintenance*, Robert Persig [Per74] comentou sobre aquilo que denominamos *qualidade*:

Qualidade... Sabemos o que ela é, embora não saibamos o que ela é. Mas essa afirmação é contraditória. Mas algumas coisas são melhores que outras; ou seja, elas têm mais qualidade. Mas quando tentamos dizer o que é qualidade, excetuando as coisas que a têm, tudo desaparece como num passe de mágica! Não há nada para dizer a respeito. Mas se não conseguimos dizer o que é Qualidade, como saber o que ela é ou como saber até se ela existe mesmo? Se ninguém sabe o que ela é, então para fins práticos ela não existiria. Porém, para fins práticos, ela realmente existe. Em que mais se baseia a qualidade? Por que outro motivo as pessoas pagariam fortunas por algumas coisas e jogariam outras na lata de lixo? Obviamente, certas coisas são melhores que outras... Mas o que é o melhor?... E por aí vai (andando em círculos), girando rodas mentais e em nenhum lugar encontrando um ponto de tração. Mas o que é mesmo Qualidade? O que é isso?

De fato — o que é isso?

Em um nível mais pragmático, David Garvin [Gar84], da Harvard Business School, sugere que “qualidade é um conceito complexo e multifacetado” que pode ser descrito segundo cinco pontos de vista diferentes. A *visão transcendental* sustenta (assim como Persig) que qualidade é algo que se reconhece imediatamente, mas não se consegue definir explicitamente. A *visão do usuário* vê a qualidade em termos das metas específicas de um usuário final. Se um produto atende a essas metas, ele apresenta qualidade. A *visão do fabricante* define qualidade em termos da especificação original do produto. Se o produto atende às especificações, ele apresenta qualidade. A *visão do produto* sugere que a qualidade pode ser ligada a características inerentes (por exemplo, funções e recursos) de um produto. Finalmente, a *visão baseada em valor* mede a qualidade tomando como base o quanto um cliente estaria disposto a pagar por um produto. Na realidade, qualidade engloba todas essas visões e outras mais.

Qualidade de projeto refere-se às características que os projetistas especificam para um produto. A qualidade dos materiais, as tolerâncias e as especificações de desempenho, todos são fatores que contribuem para a qualidade de um projeto. Quanto mais materiais de alta qualidade forem usados, tolerâncias mais rígidas e níveis de desempenho maiores forem especificados, a qualidade de projeto de um produto aumentará se o produto for fabricado de acordo com essas especificações.

No desenvolvimento de software, a qualidade de um projeto engloba o grau de atendimento às funções e características especificadas no modelo de requisitos. A *qualidade de conformidade* focaliza o grau em que a implementação segue o projeto e o sistema resultante atende suas necessidades e as metas de desempenho.



Quais são as diferentes maneiras em que a qualidade pode ser visualizada?

“As pessoas esquecem quão rápido um trabalho foi realizado — mas elas sempre lembram quão bem ele foi realizado.”

Howard Newton

Mas a qualidade do projeto e a qualidade de conformidade são as únicas questões que os engenheiros de software devem considerar? Robert Glass [Gla98] sustenta que uma relação mais “intuitiva” é o indicado:

satisfação do usuário = produto compatível + boa qualidade + entrega dentro do orçamento e do prazo previsto

Ou seja, Glass argumenta que a qualidade é importante, mas se o usuário não estiver satisfeito, nada mais importa. DeMarco [DeM98] reforça esse ponto de vista ao afirmar: “A qualidade de um produto é função do quanto ele transforma o mundo para melhor”. Essa visão de qualidade sustenta que se um produto de software fornece benefício substancial a seus usuários finais, é possível que eles estejam dispostos a tolerar problemas ocasionais de confiabilidade ou desempenho.

14.2 QUALIDADE DE SOFTWARE

Até mesmo os desenvolvedores de software mais experientes concordarão que software de alta qualidade é um objetivo importante. Mas como definir *a qualidade de software*? No sentido mais geral, a qualidade de software pode ser definida¹ como: *uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.*

Não há dúvida nenhuma de que essa definição pode ser modificada ou estendida e debatida interminavelmente. Para os propósitos deste livro, a definição serve para enfatizar três pontos importantes:

1. Uma *gestão de qualidade efetiva* estabelece a infraestrutura que dá suporte a qualquer tentativa de construir um produto de software de alta qualidade. Os aspectos administrativos do processo criam mecanismos de controle e equilíbrio de poderes que ajudam a evitar o caos no projeto — um fator-chave para uma qualidade inadequada. As práticas de engenharia de software permitem ao desenvolvedor analisar o problema e elaborar uma solução consistente —, aspectos críticos na construção de software de alta qualidade. Finalmente, as atividades de apoio como o gerenciamento de mudanças e as revisões técnicas têm muito a ver com a qualidade, assim como qualquer outra parte da prática de engenharia de software.
2. Um *produto útil* fornece o conteúdo, as funções e os recursos que o usuário final deseja, além disso, e não menos importante, deve fornecer confiabilidade e isenção de erros. Um produto útil sempre satisfaz às exigências definidas explicitamente pelos interessados. Além disso, ele satisfaz a um conjunto de requisitos implícitos (por exemplo, facilidade de uso) que se espera de todo software de alta qualidade.
3. Ao *agregar valor tanto para o fabricante quanto para o usuário* de um produto de software, um software de alta qualidade gera benefícios para a empresa de software bem como para a comunidade de usuários finais. A empresa fabricante do software ganha valor agregado pelo fato de um software de alta qualidade exigir menos manutenção, menos correções de erros e menos suporte ao cliente. Isso permite que os engenheiros de software despendam mais tempo criando aplicações novas e menos tempo em manutenções. A comunidade de usuários ganha um valor agregado, pois a aplicação fornece a capacidade de agilizar algum processo de negócio. O resultado final é: (1) maior receita gerada pelo produto de software, (2) maior rentabilidade quando uma aplicação suporta um processo de negócio, e/ou (3) maior disponibilidade de informações cruciais para o negócio.

14.2.1 Dimensões de qualidade de Garvin

David Garvin [Gar87] sugere que a qualidade deve ser considerada adotando-se um ponto de vista multidimensional que começa com uma avaliação da conformidade e termina com uma

¹ Essa definição foi adaptada de [Bes04] e substitui uma visão mais voltada para a fabricação apresentada em edições anteriores deste livro.

visão transcendental (estética). Embora as oito dimensões de qualidade de Garvin não tenham sido desenvolvidas especificamente para software, elas podem ser aplicadas quando se considera qualidade de software:

Qualidade do desempenho. O software fornece todo o conteúdo, funções e recursos que são especificados como parte do modelo de requisitos de forma a gerar valor ao usuário final?

Qualidade dos recursos. O software fornece recursos que surpreendem e encantam usuários finais que os utilizam pela primeira vez?

Confiabilidade. O software fornece todos os recursos e capacidades sem falhas? Está disponível quando necessário? Fornece funcionalidade sem a ocorrência de erros?

Conformidade. O software está de acordo com os padrões de software locais e externos relacionados com a aplicação? Segue as convenções de projeto e codificação de fato? Por exemplo, a interface com o usuário está de acordo com as regras de projeto aceitas para seleção de menus ou entrada de dados?

Durabilidade. O software pode ser mantido (modificado) ou corrigido (depurado) sem a geração involuntária de efeitos colaterais indesejados? As mudanças farão com que a taxa de erros ou a confiabilidade diminuam com o passar do tempo?

Facilidade de manutenção. O software pode ser mantido (modificado) ou corrigido (depurado) em um período de tempo aceitável e curto? O pessoal de suporte pode obter todas as informações necessárias para realizar alterações ou corrigir defeitos? Douglas Adams [Ada93] comenta ironicamente: “A diferença entre algo que pode dar errado e algo que possivelmente não pode dar errado é que quando algo que possivelmente não pode dar errado, dá errado, normalmente acaba sendo impossível acessá-lo ou repará-lo”.

Estética. Não há dúvida nenhuma de que cada um de nós tem uma visão diferente e muito subjetiva do que é estética. Mesmo assim, a maioria de nós concordaria que uma entidade estética tem certa elegância, um fluir único e uma “presença” que são difíceis de quantificar mas que, não obstante, são evidentes. Um software estético possui essas características.

Percepção. Em algumas situações, temos alguns preconceitos que influenciarão nossa percepção de qualidade. Por exemplo, se for apresentado um produto de software construído por um fornecedor que, no passado, havia produzido software de má qualidade, ficaremos com a nossa percepção de qualidade do produto de software influenciada negativamente. Similarmente, se um fornecedor tem uma excelente reputação, talvez percebamos qualidade, mesmo quando ela realmente não existe.

As dimensões de qualidade de Garvin nos dão uma visão “indulgente” da qualidade de software. Muitas (mas não todas) dessas dimensões podem ser consideradas apenas subjetivamente. Por tal razão, também precisamos de um conjunto de fatores de qualidade que podem ser classificados em duas grandes categorias: (1) fatores que podem ser medidos diretamente (por exemplo, defeitos revelados durante testes) e (2) fatores que podem ser medidos apenas indiretamente (por exemplo, usabilidade ou facilidade de manutenção). Em cada um dos casos deve ocorrer medição. Devemos comparar o software com algum dado e chegarmos a uma indicação da qualidade.

14.2.2 Fatores de qualidade de McCall

McCall, Richards e Walters [McC77] criaram uma proposta de categorização dos fatores que afetam a qualidade de software. Esses fatores de qualidade de software, apresentados na Figura 14.1, focam em três importantes aspectos de um produto de software: as características operacionais, a habilidade de suportar mudanças e a adaptabilidade a novos ambientes.

Referindo-se aos fatores citados na Figura 14.1, McCall e seus colegas fazem as seguintes descrições:

Correção. O quanto um programa satisfaz a sua especificação e atende aos objetivos da missão do cliente.

Confiabilidade. O quanto se pode esperar que um programa realize a função pretendida com a precisão exigida. [Observe que foram propostas outras definições mais completas de confiabilidade (veja o Capítulo 25).]

Eficiência. A quantidade de recursos computacionais e código exigidos por um programa para desempenhar sua função.

Integridade. O quanto o acesso ao software ou dados por pessoas não autorizadas pode ser controlado.

Usabilidade. Esforço necessário para aprender, operar, preparar a entrada de dados e interpretar a saída de um programa.

Facilidade de manutenção. Esforço necessário para localizar e corrigir um erro em um programa. [Trata-se de uma definição muito limitada.]

Flexibilidade. Esforço necessário para modificar um programa em operação.

Testabilidade. Esforço necessário para testar um programa de modo a garantir que ele desempenhe a função destinada.

Portabilidade. Esforço necessário para transferir o programa de um ambiente de hardware e/ou software para outro.

Reusabilidade. O quanto um programa [ou partes de um programa] pode ser reutilizado em outras aplicações — relacionado com o empacotamento e o escopo das funções que o programa executa.

Interoperabilidade. Esforço necessário para integrar um sistema a outro.

É difícil, e em alguns casos impossível, desenvolver medidas² diretas desses fatores de qualidade. Na realidade, muitas das métricas definidas por McCall et al. podem ser medidas apenas indiretamente. Entretanto, avaliar a qualidade de uma aplicação usando esses fatores possibilitará uma sólida indicação da qualidade de um software.

"O amargo da má qualidade permanece muito tempo depois da doçura do cumprimento do prazo ter sido esquecida."

Karl Weigers
(citação não atribuída)

FIGURA 14.1

Fatores de qualidade de software de McCall



14.2.3 Fatores de qualidade ISO 9126

O padrão ISO 9126 foi desenvolvido como uma tentativa de identificar os atributos fundamentais de qualidade para software de computador. O padrão identifica seis atributos fundamentais de qualidade:

² Uma *medida direta* implica existir um único valor contável que dê uma indicação direta do atributo que está sendo examinado. Por exemplo, o "tamanho" de um programa pode ser medido diretamente contando-se o número de linhas de código.



Embora seja tentador criar medidas quantitativas para os fatores de qualidade aqui citados, também podemos criar uma lista de verificação simples dos atributos que dão uma sólida indicação de que o fator está presente.

Funcionalidade. O grau com que o software satisfaz às necessidades declaradas conforme indicado pelos seguintes subatributos: adequabilidade, exatidão, interoperabilidade, conformidade e segurança.

Confiabilidade. A quantidade de tempo que o software fica disponível para uso conforme indicado pelos seguintes subatributos: maturidade, tolerância a falhas, facilidade de recuperação.

Usabilidade. O grau de facilidade de utilização do software conforme indicado pelos seguintes subatributos: facilidade de compreensão, facilidade de aprendizagem, operabilidade.

Eficiência. O grau de otimização do uso, pelo software, dos recursos do sistema conforme indicado pelos seguintes subatributos: comportamento em relação ao tempo, comportamento em relação aos recursos.

Facilidade de manutenção. A facilidade com a qual uma correção pode ser realizada no software conforme indicado pelos seguintes subatributos: facilidade de análise, facilidade de realização de mudanças, estabilidade, testabilidade.

Portabilidade. A facilidade com a qual um software pode ser transposto de um ambiente a outro conforme indicado pelos seguintes subatributos: adaptabilidade, facilidade de instalação, conformidade, facilidade de substituição.

Assim como outros fatores de qualidade de software discutidos nas subseções anteriores, os fatores da ISO 9126 não levam, necessariamente, à medição direta. Entretanto, eles fornecem uma base razoável para medidas indiretas e uma excelente lista de verificação para avaliar a qualidade de um sistema.

14.2.4 Fatores de qualidade desejados

As dimensões e os fatores de qualidade apresentados nas Seções 14.2.1 e 14.2.2 concentram-se no software como um todo e podem ser usados como uma indicação genérica da qualidade de uma aplicação. A equipe de software pode desenvolver um conjunto de características de qualidade e questões associadas que investigaria³ o grau em que cada fator foi satisfeito. Por exemplo, McCall identifica a *usabilidade* como sendo um importante fator de qualidade. Se lhe fosse solicitado para revisar uma interface com o usuário e avaliar sua usabilidade, como você procederia? Você poderia começar com os subatributos sugeridos por McCall — facilidade de compreensão, facilidade de aprendizagem, operabilidade — mas qual seu significado em um sentido prático?

Para conduzir sua avaliação, você precisaria lidar com atributos específicos, mensuráveis (ou pelo menos, reconhecíveis) da interface. Por exemplo [Bro03]:

Intuição. O grau em que a interface segue padrões de uso esperados de modo que até mesmo um novato possa usá-la sem treinamento significativo.

- O layout da interface favorece a fácil compreensão?
- As operações da interface são fáceis de ser localizadas e iniciadas?
- A interface utiliza uma metáfora reconhecível?
- É especificada entrada para economizar toques de teclado ou cliques de mouse?
- A interface segue as três regras de ouro? (Ver o Capítulo 11.)
- A estética ajuda no entendimento e uso?

Eficiência. A facilidade com a qual as operações e informações podem ser localizadas ou iniciadas.

- O layout e o estilo da interface permitem a um usuário localizar eficientemente as operações e informações?

“Qualquer atividade se torna criativa quando o executor se preocupa em fazê-la de maneira correta ou melhor.”

John Updike

³ Essas características e questões podem ser abordadas como parte de uma revisão de software (Capítulo 15).

- Uma sequência de operações (ou entrada de dados) pode ser realizada reduzindo-se o número de movimentos?
- Os dados de saída ou o conteúdo são apresentados de modo a ser imediatamente compreendidos?
- As operações hierárquicas foram organizadas de modo a minimizar o nível em que um usuário deve navegar para realizar algo?

Robustez. O grau com o qual o software trata dados incorretos de entrada ou interação inapropriada com o usuário.

- O software reconhecerá erros caso sejam introduzidos dados dentro ou fora dos limites prescritos? Mais importante ainda, o software continuará a operar sem falha ou degradação?
- A interface reconhece erros cognitivos ou manipuladores comuns e orienta explicitamente o usuário para retomar o caminho certo?
- A interface oferece diagnósticos e orientação úteis quando é descoberta uma condição de erro (associada à funcionalidade do software)?

Riqueza. O grau em que a interface oferece um conjunto rico de recursos importantes.

- A interface pode ser personalizada de acordo com as necessidades específicas de um usuário?
- A interface dispõe de recursos de macros que permitem ao usuário identificar uma sequência de operações comuns por meio de uma única ação ou comando?

Enquanto o projeto de interface é desenvolvido, a equipe de software revisaria o protótipo de projeto e faria as perguntas citadas. Se a resposta a essas perguntas for “sim”, é provável que a interface com o usuário apresente alta qualidade. Um conjunto de perguntas similares deveria ser desenvolvido para cada fator de qualidade a ser avaliado.

14.2.5 A transição para uma visão quantitativa

Nas subseções anteriores, apresentamos diversos fatores qualitativos para a “medição” da qualidade de um software. A comunidade da engenharia de software se esforça ao máximo para desenvolver medidas precisas para a qualidade de software e algumas vezes é frustrada pela natureza subjetiva da atividade. Cavano e McCall [Cav78] discutem essa situação:

A determinação da qualidade é um fator-chave nos eventos do dia a dia — concursos de degustação de vinhos, eventos esportivos (por exemplo, ginástica), concursos de talentos etc. Nessas situações, a qualidade é julgada da forma mais fundamental e direta: comparação entre dois objetos sob condições idênticas e com conceitos predeterminados. O vinho pode ser julgado segundo sua limpidez, cor, buquê, sabor etc. Entretanto, esse tipo de julgamento é muito subjetivo; para ter algum valor, ele precisa ser feito por um especialista.

A subjetividade e a especialização também se aplicam na determinação da qualidade do software. Para ajudar a solucionar esse problema, é preciso uma definição mais precisa da qualidade de software bem como uma maneira de obter medidas quantitativas da qualidade de software para uma análise objetiva... Como não existe conhecimento absoluto sobre isso, não se deve ter a expectativa de medir a qualidade de software com exatidão, já que cada medida é parcialmente imperfeita. Jacob Bronkowsky descreve esse paradoxo do conhecimento da seguinte maneira: “Ano após ano, inventamos instrumentos mais precisos com os quais observamos a natureza com maior perfeição. E ao analisarmos essas observações, ficamos desconcertados em ver que elas ainda são distorcidas e ainda incertas”.

No Capítulo 23, apresentaremos um conjunto de métricas de software que podem ser aplicadas para a avaliação quantitativa da qualidade de software. Em todos os casos, as métricas representam medidas indiretas; isto é, jamais medimos realmente a *qualidade*, mas sim alguma manifestação dessa qualidade. O fator complicador é a relação precisa entre a variável medida e a qualidade de software.

14.3 O DILEMA DA QUALIDADE DE SOFTWARE



Ao deparar com o dilema da qualidade (e todo mundo depara com ele uma hora ou outra), tente alcançar o equilíbrio — esforço suficiente para produzir qualidade aceitável sem “enterrar” o projeto.

Em uma entrevista [Ven03] publicada na Internet, Bertrand Meyer discute o que chamamos de *dilema da qualidade*:

Se produzimos um sistema de software de péssima qualidade, perdemos porque ninguém irá querer comprá-lo. Se, por outro lado, gastamos um tempo infinito, um esforço extremamente grande e grandes somas de dinheiro para construir um software absolutamente perfeito, então isso levará muito tempo para ser completado, e o custo de produção será tão alto que iremos à falência. Ou perdemos a oportunidade de mercado ou então simplesmente esgotamos todos os nossos recursos. Dessa maneira, os profissionais desta área tentam encontrar aquele meio-termo mágico onde o produto é suficientemente bom para não ser rejeitado logo de cara, como, por exemplo, durante uma avaliação, mas também não é o objeto de tamanho perfeccionismo e trabalho que levaria muito tempo ou que custaria demasiadamente para ser finalizado.

É válido afirmar que os engenheiros de software devem se esforçar para produzir sistemas de alta qualidade. Muito melhor seria aplicar boas práticas na tentativa de obter essa alta qualidade. Porém, a situação discutida por Meyer é a realidade e representa um dilema até mesmo para as melhores organizações de engenharia de software.

14.3.1 Software “bom o suficiente”

Falando claramente, se devemos aceitar o argumento feito por Meyer, é aceitável produzirmos software “bom o suficiente”? A resposta a essa pergunta deve ser “sim”, pois atualmente, as principais empresas de software agem dessa forma. Elas criam software com erros (bugs) conhecidos e os entregam a vários usuários finais. Elas reconhecem que algumas funções e características disponibilizadas na Versão 1.0 podem não ser da melhor qualidade e planejam melhoramentos para a Versão 2.0. Elas fazem isso mesmo sabendo que alguns clientes irão reclamar, entretanto, reconhecem que o tempo de colocação do produto no mercado é a melhor cartada de qualidade desde que o produto fornecido seja “bom o suficiente”.

O que é exatamente “bom o suficiente”? Software bom o suficiente fornece funções e características de alta qualidade que os usuários desejam, mas, ao mesmo tempo, fornece outras funções e características mais obscuras ou especializadas e ainda contendo erros conhecidos. O fornecedor de software espera que a grande maioria dos usuários ignore os erros pelo fato de estarem muito satisfeitos com as outras funcionalidades oferecidas pela aplicação.

Essa ideia pode ter um significado especial para muitos leitores. Se você for um deles, pedimos para considerar alguns dos argumentos contra software “bom o suficiente”.

É verdade que software “bom o suficiente” pode funcionar em alguns domínios de aplicação e para algumas grandes empresas de software. Afinal de contas, se uma empresa tiver um grande orçamento para o marketing e conseguir convencer um número suficiente de pessoas a comprar a versão 1.0, ela será bem-sucedida. Conforme citado anteriormente, pode-se argumentar que a qualidade será melhorada nas próximas versões. Ao entregar a versão 1.0 boa o suficiente, a empresa já monopolizou o mercado.

Caso trabalhe em uma pequena empresa, não confie nessa filosofia. Ao entregar um produto bom o suficiente (com erros), você corre o risco de arruinar permanentemente a reputação da empresa. Talvez jamais tenha a chance de entregar a versão 2.0 pois, devido à má propaganda resultante dessa filosofia, as vendas podem despencar e, conseqüentemente, a empresa falir.

Caso trabalhe em certos domínios de aplicação (por exemplo, software embarcado para aplicações em tempo real) ou crie software de aplicação integrado com o hardware (por exemplo, software para a indústria automotiva, software para telecomunicações), entregar software com erros conhecidos pode ser negligente e expõe sua empresa a litígios custosos. Em alguns casos, pode constituir crime. Ninguém quer software bom o suficiente e inútil!

Portanto, proceda com cautela caso acredite que “bom o suficiente” seja um atalho capaz de resolver seus problemas de qualidade de software. Pode ser que funcione, mas apenas para poucos casos e em um conjunto limitado de domínios de aplicação.⁴

14.3.2 Custo da qualidade

A discussão prossegue mais ou menos assim — *sabemos que a qualidade é importante, mas ela nos custa tempo e dinheiro — tempo e dinheiro em demasia para obter o nível de qualidade de software que realmente desejamos*. Dessa forma, o argumento parece razoável (veja os comentários de Meyer no início desta seção). Não há dúvida nenhuma de que a qualidade tem um preço, mas a falta de qualidade também tem um preço — não apenas para os usuários finais, que terão de conviver com um software com erros, mas também para a organização de software que o criou e, além de tudo, terá de fazer a manutenção. A verdadeira questão é a seguinte: *com qual custo deveríamos nos preocupar?* Para responder a essa pergunta, devemos entender tanto o custo para atingir alta qualidade quanto o custo de software de baixa qualidade.

O *custo da qualidade* inclui todos os custos necessários para a busca de qualidade ou para a execução de atividades relacionadas à qualidade, assim como os custos causados pela falta de qualidade. Para compreender esses custos, uma organização deve reunir métricas para prover uma base para o custo corrente da qualidade, identificar oportunidades para reduzir esses custos e fornecer uma base normalizada de comparação. O custo da qualidade pode ser dividido em custos associados à prevenção, avaliação e falhas.

Os *custos de prevenção* incluem: (1) o custo de atividades de gerenciamento necessárias para planejar e coordenar todas as atividades de controle e garantia da qualidade, (2) o custo de atividades técnicas adicionais para desenvolver modelos completos de requisitos e de projeto, (3) custos de planejamento de testes e (4) o custo de todo o treinamento associado a essas atividades.

Os *custos de avaliação* incluem atividades para a compreensão aprofundada da condição do produto “a primeira vez através de” cada processo. Entre os exemplos de custos de avaliação, temos:

- Custo para realização de revisões técnicas (Capítulo 15) para produtos resultantes de engenharia de software.
- Custo para coleta de dados e avaliação de métricas (Capítulo 23).
- Custo para testes e depuração (Capítulos 18 a 21).

Os *custos de falhas* são aqueles que desapareceriam caso nenhum erro tivesse surgido antes ou depois da entrega de um produto a clientes. Esses custos podem ser subdivididos em custos de falhas internas e custos de falhas externas. Os custos de falhas internas ocorrem quando se detecta um erro em um produto antes de ele ser entregue e abrangem:

- Custo necessário para realizar retrabalhos (reparos) para corrigir um erro.
- Custo que ocorre quando retrabalhos geram, inadvertidamente, efeitos colaterais que devem ser reduzidos.
- Custos associados à reunião de métricas de qualidade que permitem a uma organização avaliar os modos de falha.

Os *custos de falhas externas* estão associados a defeitos encontrados após o produto ter sido entregue ao cliente. Exemplos de custos de falhas externas são resolução de reclamações, devolução e substituição de produtos, suporte telefônico/via e-mail e custos de mão de obra associados à garantia do produto. Uma má reputação e a consequente perda de negócios é outro custo de falhas externas difícil de ser quantificado, mas bastante real. Coisas negativas acontecem quando se produz software de baixa qualidade. Em uma censura aos



Não tenha medo de incorrer em custos significativos de prevenção. Esteja seguro que esse investimento possibilitará um excelente retorno.

“Leva menos tempo para fazer algo certo do que explicar porque o fez errado.”

H. W.
Longfellow

⁴ Uma interessante discussão dos prós e contras de software “bom o suficiente” pode ser encontrada em [Bre02].

desenvolvedores de software que se recusam a considerar os custos de falhas externas, Cem Kaner [Kan95] afirma:

Muitos dos custos de falhas externas, assim como sua reputação no mercado, são difíceis de ser quantificados e, conseqüentemente, muitas empresas os ignoram no cálculo das relações custo-benefício. Outros custos de falhas externas podem ser reduzidos (por exemplo, o fornecimento de suporte pós-venda mais barato e de menor qualidade ou cobrando o suporte dos clientes) sem aumentar a satisfação do cliente. Ao ignorar os custos de produtos ruins para nossos clientes, os engenheiros de qualidade encorajam tomadas de decisão relacionadas à qualidade que penalizam nossos clientes em vez de satisfazê-los.

Como era de esperar, os custos relativos para descobrir e reparar um erro ou defeito aumentam drasticamente à medida que avançamos dos custos de prevenção para custos de detecção de falhas internas e para custos de falhas externas. A Figura 14.2, fundamentada em dados coletados por Boehm e Basili [Boe01b] e ilustrada pela Cigital Inc. [Cig07], exemplifica este fenômeno.

O custo médio da indústria de software para corrigir um defeito durante a geração de código é aproximadamente US\$ 977 por erro. O custo médio para corrigir o mesmo erro caso ele tenha sido descoberto durante os testes do sistema passa a ser de US\$ 7.136 por erro. A Cigital Inc. [Cig07] considera uma grande aplicação em que foram introduzidos 200 erros durante a codificação:

De acordo com os dados médios do setor, o custo para descobrir e corrigir defeitos durante a fase de codificação é de US\$ 977 por defeito. Portanto, o custo total para corrigir os 200 defeitos “críticos” durante essa fase ($200 \times \text{US\$ } 977$) é de aproximadamente US\$ 195.400.

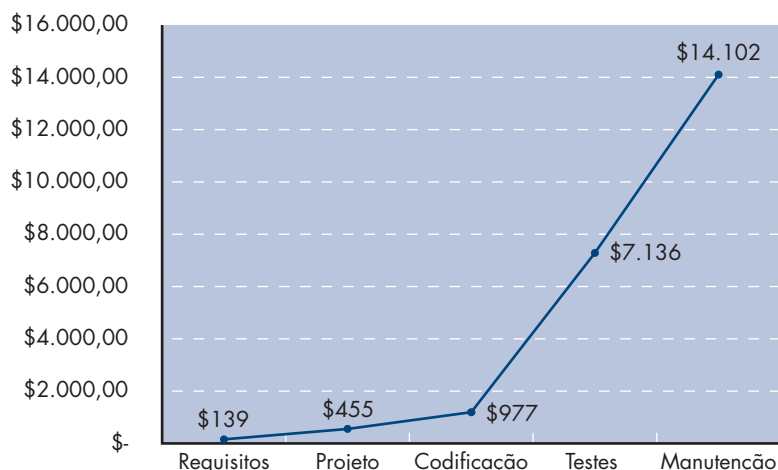
Os dados médios do setor mostram que o custo para descobrir e corrigir defeitos durante a fase de testes é de US\$ 7.136 por defeito. Nesse caso, supondo que a fase de testes do sistema tenha revelado aproximadamente 50 defeitos críticos (ou apenas 25% daqueles encontrados pela Cigital na fase de codificação), o custo para descobrir e corrigir esses defeitos ($50 \times \text{US\$ } 7.136$) teria sido de aproximadamente US\$ 356.800. Isso teria resultado em 150 erros críticos sem serem detectados e corrigidos. O custo para descobrir e corrigir esses 150 defeitos remanescentes na fase de manutenção ($150 \times \text{US\$ } 14.102$) teria sido de US\$ 2.115.300. Portanto, o custo total para descobrir e corrigir os 200 defeitos após a fase de codificação teria sido de US\$ 2.472.100 ($\text{US\$ } 2.115.300 + \text{US\$ } 356.800$).

Mesmo que sua organização de software possua custos equivalentes à metade da média do setor (a maioria não tem a mínima ideia de quanto são seus custos!), a economia de custos associados a atividades iniciais de controle com garantia da qualidade (conduzidas durante a análise de requisitos e projeto) é considerável.

FIGURA 14.2

**Custo relativo para
correção de erros
e defeitos**

Fonte: Adaptado de
(Boe01b)



14.3.3 Riscos

No Capítulo 1 foi escrito que “as pessoas apostam seus empregos, comodidades, segurança, entretenimento, decisões e as próprias vidas em software. Esperamos que elas estejam certas”. A implicação disso é que software de baixa qualidade aumenta os riscos tanto para o desenvolvedor quanto para o usuário final. Na subseção anterior, discutimos um desses riscos (custo). Mas o lado negativo de aplicações mal projetadas e implementadas nem sempre resulta apenas em altos custos e mais tempo. Um exemplo [Gag04] extremo pode servir como ilustração.

Ao longo do mês de novembro de 2000, em um hospital no Panamá, 28 pacientes receberam doses maciças de raio gama durante tratamento para uma série de tipos de câncer. Nos meses seguintes, 5 desses pacientes morreram por contaminação radioativa e outros 15 desenvolveram sérias complicações. O que provocou essa tragédia? Um pacote de software, desenvolvido por uma companhia americana, foi modificado por técnicos do hospital para calcular doses de radiação para cada paciente.

Os três médicos panamenhos com especialização em física, que “ajustaram” o software para aumentar a capacidade funcional, foram processados por homicídio doloso, mas sem premeditação. A empresa americana está enfrentando litígios graves em dois países. Gage e McCormick comentam:

Não se trata de um conto com fundo moral para técnicos em medicina, muito embora eles possam lutar para manter-se fora da prisão caso tenham interpretado ou usado uma tecnologia de forma errada. Não se trata também de um conto de como seres humanos podem ser feridos ou sofrer algo ainda mais grave por software mal projetado ou documentado, embora existam muitos exemplos que o comprovem. Trata-se de um alerta para qualquer criador de programas de computador: a qualidade de software é importante, as aplicações têm de ser infalíveis e — independentemente de estarem embutidas no motor de um carro, em um braço mecânico em uma fábrica ou em um aparelho médico em um hospital — código mal empregado pode matar.

A baixa qualidade induz a riscos, alguns muito sérios.

14.3.4 Negligência e responsabilidade civil

A história é bastante comum. Um órgão do governo ou empresa contrata uma grande consultoria ou uma importante empresa desenvolvedora de software para analisar os requisitos e então projetar e construir um “sistema” baseado em software para apoiar alguma atividade importante. O sistema poderia oferecer suporte a uma importante função corporativa (por exemplo, administração de aposentadorias) ou alguma função governamental (por exemplo, administração do sistema de saúde ou de segurança do território nacional).

O trabalho inicia com as melhores das intenções de ambas as partes, mas na época em que o sistema é entregue, as coisas não andam bem. O sistema é lento, não fornece os recursos e funções desejados, é suscetível a erros e não tem a aprovação do usuário. Segue-se um litígio. Na maioria dos casos, o cliente alega que o desenvolvedor foi negligente (na maneira de aplicação de práticas de software) e, portanto, não tem direito a receber seu pagamento. Normalmente, o desenvolvedor alega que o cliente mudou repetidamente os requisitos e subverteu a parceria de desenvolvimento de outras formas. Em todos os casos, a questão é a qualidade do sistema entregue.

14.3.5 Qualidade e segurança

À medida que a criticabilidade das aplicações e dos sistemas baseados na Internet aumenta, a segurança da aplicação tem se tornando cada vez mais importante. Ou seja, software que não apresente alta qualidade é mais fácil de ser copiado (“pirateado”) e, como consequência, o software de baixa qualidade pode aumentar indiretamente o risco de segurança assim como todos os problemas e custos associados.

Numa entrevista à revista *ComputerWorld*, o autor e especialista em segurança Gary McGraw comenta [Wil05]:

A segurança de software se relaciona inteira e completamente à qualidade. Devemos nos preocupar com a segurança, a confiabilidade, a disponibilidade e a dependência — nas fases inicial, de projeto, de arquitetura, de testes e de codificação, ao longo de todo o ciclo de vida [qualidade] de um software. Até mesmo pessoas cientes do problema de segurança têm se concentrado em coisas relacionadas ao ciclo de vida do software. O quanto antes descobrirmos um problema de software, melhor. E existem dois tipos de problemas de software. O primeiro são os bugs, que são problemas de implementação. Os demais são falhas de software — problemas arquiteturais do projeto. As pessoas prestam muita atenção aos bugs, mas não o suficiente às falhas.

Para construir um sistema seguro, temos de focar na qualidade, e esse foco deve iniciar durante o projeto. Os conceitos e os métodos discutidos na Parte 2 deste livro nos conduzem a uma arquitetura de software que reduz “falhas”. Ao eliminar falhas arquiteturais (melhorando, consequentemente, a qualidade do software), faremos com que fique muito mais difícil a cópia ilegal do software.

14.3.6 O impacto das ações administrativas

A qualidade de software é normalmente tão influenciada pelas decisões administrativas quanto pelas decisões técnicas. Até mesmo as melhores práticas de engenharia de software podem ser subvertidas por decisões de negócios inadequadas e ações questionáveis de gerenciamento de projeto.

Na Parte 4 deste livro discutiremos o gerenciamento de projetos no contexto da gestão da qualidade. Quando cada tarefa de projeto é iniciada, um líder de projeto tomará decisões que podem ter um impacto significativo sobre a qualidade do produto.

Decisões de estimativas. Conforme citamos no Capítulo 26, raramente uma equipe de software pode se dar ao luxo de fornecer uma estimativa para um projeto *antes* das datas de entrega serem estabelecidas e um orçamento geral ser especificado. Em vez disso, a equipe realiza um “exame de sanidade” para garantir que as datas de entrega são racionais. Em muitos casos, existe uma enorme pressão de colocação do produto no mercado que força uma equipe a aceitar as datas impraticáveis de entrega. Como resultado, são realizados atalhos, as atividades que produzem um software de maior qualidade talvez sejam deixadas de lado e, assim, a qualidade do produto sofre as consequências. Se uma data de entrega for absurda, é importante manter-se firme. Explique por que você precisa de mais tempo ou, alternativamente, sugira um subconjunto de funcionalidades que possa ser entregue (com alta qualidade) no tempo alocado.

Decisões de cronograma. Quando um cronograma de projeto de software é estabelecido (Capítulo 27), as tarefas são sequenciadas tomando-se como base as dependências. Por exemplo, como o componente **A** depende do processamento que ocorre nos componentes **B**, **C** e **D**, o componente **A** não pode ser agendado para testes até que os componentes **B**, **C** e **D** sejam completamente testados. O cronograma de um projeto deve refletir essa situação. Mas se o tempo for muito curto e **A** tem de estar disponível para outros testes críticos, talvez se opte por testar **A** sem seus componentes subordinados (que estão ligeiramente atrasados em relação ao cronograma), de modo a torná-lo disponível para outros testes que devem ser feitos antes da entrega. Afinal de contas, o prazo final se aproxima. Dessa forma, **A** pode ter defeitos que estão ocultos e que seriam descobertos muito mais tarde. A qualidade sofre as consequências.

Decisões orientadas a riscos. A administração de riscos (Capítulo 28) é um dos atributos fundamentais de um projeto de software bem-sucedido. Precisamos realmente saber o que poderia dar errado e estabelecer um plano de contingência caso isso aconteça. Um número muito grande de equipes de software prefere um otimismo cego, estabelecendo um cronograma de desenvolvimento sob a hipótese de que nada vai dar errado. Pior ainda, eles não têm um método para lidar com as coisas que dão errado. Consequentemente, quando um risco se torna realidade, reina o caos e, à medida que o grau de loucura aumenta, o nível de qualidade invariavelmente cai.

O dilema da qualidade de software pode ser mais bem sintetizado enunciando-se a Lei de Meskimen — *Nunca há tempo para fazer a coisa certa, mas sempre há tempo para fazê-la de novo*. Meu conselho: tomar o tempo necessário para fazer certo da primeira vez quase nunca é uma decisão errada.

14.4 ALCANÇANDO A QUALIDADE DE SOFTWARE

A qualidade de software não aparece simplesmente do nada. Ela é o resultado de um bom gerenciamento de projeto e uma prática consistente de engenharia de software. O gerenciamento e a prática são aplicados no contexto de quatro grandes atividades que ajudam uma equipe de software a atingir alto padrão de qualidade de software: métodos de engenharia de software, técnicas de gerenciamento de projeto, ações de controle de qualidade e garantia da qualidade de software.

14.4.1 Métodos de engenharia de software

Se nossa expectativa é construir software de alta qualidade, temos de entender o problema a ser resolvido. Temos também de ser capazes de criar um projeto que seja adequado ao problema e, ao mesmo tempo, apresente características que levem a um software com as dimensões e fatores de qualidade discutidos na Seção 14.2.

Na Parte 2 deste livro, apresentamos uma ampla gama de conceitos e métodos capazes de levar a um entendimento relativamente completo do problema e a um projeto abrangente que estabeleça uma base sólida para a atividade de construção. Se aplicarmos esses conceitos e adotarmos os métodos de análise e projeto apropriados, a probabilidade de criarmos software de alta qualidade aumentará substancialmente.

14.4.2 Técnicas de gerenciamento de software

O impacto de decisões de gerenciamento inadequadas sobre a qualidade de software foi discutido na Seção 14.3.6. As implicações são claras: se (1) um gerente de projeto usar estimativas para verificar que as datas de entrega são plausíveis, (2) as dependências de cronograma forem entendidas e a equipe resistir à tentação de usar atalhos, (3) o planejamento de riscos for conduzido de modo que problemas não gerem caos, a qualidade do software será afetada de forma positiva.

Além disso, o plano de projeto deve incluir técnicas explícitas para gerenciamento de mudanças e qualidade. Técnicas que levam a práticas ótimas de gerenciamento de projeto são discutidas na Parte 4 do livro.

14.4.3 Controle de qualidade

O controle de qualidade engloba um conjunto de ações de engenharia de software que ajudam a garantir que cada produto resultante atinja suas metas de qualidade. Os modelos são revistos de modo a garantir que sejam completos e consistentes. O código poderia ser inspecionado de modo a revelar e corrigir erros antes de os testes começarem. Aplica-se uma série de etapas de teste para descobrir erros na lógica de processamento, na manipulação de dados e na comunicação da interface. Uma combinação de medições e realimentação (feedback) permite a uma equipe de software ajustar o processo quando qualquer um desses produtos resultantes deixe de atender às metas estabelecidas para a qualidade. As atividades de controle de qualidade são discutidas em detalhe ao longo do restante da Parte 3 deste livro.

14.4.4 Garantia da qualidade

A garantia da qualidade estabelece a infraestrutura que suporta métodos sólidos de engenharia de software, gerenciamento racional de projeto e ações de controle de qualidade — todos fundamentais para a construção de software de alta qualidade. Além disso, a garantia

 O que é preciso fazer para afetar a qualidade de forma positiva?

 O que é controle de qualidade de software?

WebRef

Links úteis para recursos sobre garantia da qualidade de software podem ser encontrados em www.niwotridge.com/Resources/PMSWEResources/SoftwareQualityAssurance.htm.

da qualidade consiste em um conjunto de funções de auditoria e de relatórios que possibilita uma avaliação da efetividade e da completude das ações de controle de qualidade. O objetivo da garantia da qualidade é fornecer ao pessoal técnico e administrativo os dados necessários para ser informados sobre a qualidade do produto, ganhando, portanto, entendimento e confiança de que as ações para atingir a qualidade desejada do produto estão funcionando. Obviamente, se os dados fornecidos pela garantia da qualidade identificarem problemas, é responsabilidade do gerenciamento tratar esses problemas e aplicar os recursos necessários para resolver os problemas de qualidade. A garantia da qualidade de software é discutida em detalhe no Capítulo 16.

14.5 RESUMO

A preocupação com a qualidade de sistemas de software cresceu à medida que o software passou a se tornar cada vez mais integrado em cada aspecto da vida cotidiana. Mas é difícil desenvolver uma descrição completa sobre qualidade de software. Neste capítulo, a qualidade foi definida como uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.

Foi proposta uma grande variedade de dimensões e fatores para qualidade de software ao longo dos anos. Todos tentam definir um conjunto de características que, se atingidas, levarão a um software de alta qualidade. Os fatores de qualidade de McCall e da ISO 9126 estabelecem características como confiabilidade, usabilidade, facilidade de manutenção, funcionalidade e portabilidade como indicadores de que a qualidade existe.

Todas as organizações envolvidas com software deparam com o dilema da qualidade de software. Em essência, todos querem construir sistemas de alta qualidade, mas o tempo e o esforço necessários para produzir um software “perfeito” simplesmente não existem em um mundo orientado ao mercado. A pergunta passa a ser: devemos construir software “bom o suficiente”? Embora muitas empresas façam exatamente isso, há um grande porém que deve ser considerado.

Independentemente da abordagem escolhida, a qualidade tem, efetivamente, um custo que pode ser discutido em termos de prevenção, avaliação e falha. Os custos de prevenção incluem todas as ações de engenharia de software que são desenvolvidas para, em primeiro lugar, evitar defeitos. Os custos de avaliação estão associados àquelas ações que avaliam os artefatos resultantes para determinar sua qualidade. Os custos de falhas englobam o preço de falhas internas e os efeitos externos que a má qualidade gera.

A qualidade de software é atingida por meio da aplicação de métodos de engenharia de software, práticas administrativas consistentes e controle de qualidade completo — todos suportados por uma infraestrutura de garantia da qualidade de software. Nos capítulos seguintes, o controle e a garantia da qualidade são discutidos com maior nível de detalhamento.

PROBLEMAS E PONTOS A PONDERAR

14.1. Descreva como você avaliaria a qualidade de uma universidade antes de se candidatar a ela. Quais fatores seriam importantes? Quais seriam críticos?

14.2. Garvin [Gar84] descreve cinco visões diferentes de qualidade. Dê um exemplo de cada uma delas usando um ou mais produtos eletrônicos conhecidos com os quais você esteja familiarizado.

14.3. Usando a definição de qualidade de software proposta na Seção 14.2, você acredita que seja possível criar um produto útil que gere valor mensurável sem usar um processo eficaz? Justifique sua resposta.

14.4. Acrescente duas outras perguntas a cada uma das dimensões de qualidade de Garvin apresentadas na Seção 14.2.1.

14.5. Os fatores de qualidade de McCall foram desenvolvidos durante a década de 1970. Praticamente todos os aspectos da computação mudaram drasticamente desde essa época e, mesmo assim, os fatores de McCall ainda se aplicam a software moderno. Você seria capaz de tirar alguma conclusão com base nesse fato?

14.6. Usando os subatributos citados para o fator de qualidade da ISO 9126, “facilidade de manutenção”, na Seção 14.2.3, desenvolva um conjunto de perguntas que explore se esses atributos estão presentes ou não. Siga o exemplo mostrado na Seção 14.2.4.

14.7. Descreva o dilema da qualidade de software com suas próprias palavras.

14.8. O que é software “bom o suficiente”? Cite uma empresa específica e produtos específicos que você acredita terem sido desenvolvidos usando essa filosofia.

14.9. Considerando cada um dos quatro aspectos do custo da qualidade, qual você acredita ser o mais caro e a razão para sua resposta?

14.10. Faça uma busca na Internet e encontre três outros exemplos de “riscos” para o público que podem ser diretamente atribuídos à baixa qualidade de software. Considere a opção de iniciar sua pesquisa em <http://catless.ncl.ac.uk/risks>.

14.11. *Qualidade e segurança* são a mesma coisa? Explique.

14.12. Explique por que muitos de nós continuamos a viver da lei de Meskimen. Como isso se aplica a um negócio de software?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Conceitos básicos sobre qualidade de software são considerados em livros como os de Henry e Hanlon (*Software Quality Assurance*, Prentice-Hall, 2008), Khan e seus colegas (*Software Quality: Concepts and Practice*, Alpha Science International, Ltd., 2006), O'Regan (*A Practical Approach to Software Quality*, 2002) e Daughtrey (*Fundamental Concepts for the Software Quality Engineer*, ASQ Quality Press, 2001).

Duvall e seus colegas (*Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), Kandt (*Software Engineering Quality Practices*, Auerbach, 2005), Godbole (*Software Quality Assurance: Principles and Practice*, Alpha Science International, Ltd., 2004) e Galin (*Software Quality Assurance: From Theory to Implementation*, Addison-Wesley, 2003) apresentam tratados detalhados sobre SQA. A garantia de qualidade no contexto do processo ágil é considerada por Stamelos e Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007).

Projeto consistente leva a qualidade de software elevada. Jayasawal e Patton (*Design for Trustworthy Software*, Prentice-Hall, 2006) e Ploesch (*Contracts, Scenarios and Prototypes*, Springer, 2004) discutem ferramentas e técnicas para desenvolver software “robusto”.

A medição é um importante componente da engenharia de qualidade de software. Ejiogu (*Software Metrics: The Discipline of Software Quality*, BookSurge Publishing, 2005), Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2002) e Nance e Arthur (*Managing Software Quality*, Springer, 2002) discutem importantes métricas e modelos relacionados com a qualidade. Os aspectos da qualidade de software orientados a equipes são considerados por Evans (*Achieving Software Quality through Teamwork*, Artech House Publishers, 2004).

Uma ampla gama de fontes de informação sobre qualidade de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes para a qualidade de software pode ser encontrada no site www.mhhe.com/engcs/compSCI/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

amplificação de	
defeitos	375
defeitos	374
densidade de erros	377
erros	374
manutenção de	
registros	382
revisão	
métrica	376
relatório	382
revisões	
eficácia de custos	377
informais	380
por amostragem.	384
técnicas	381

As revisões de software são como um “filtro” para a gestão de qualidade. Isso significa que as revisões são aplicadas em várias etapas durante o processo de engenharia de software e servem para revelar erros e defeitos que podem ser eliminados. As revisões de software “purificam” o resultado do trabalho da engenharia de software, até mesmo os modelos de requisitos e de projeto, dados de teste e código. Freedman e Weinberg [Fre90] discutem a necessidade de revisões da seguinte maneira:

O trabalho técnico precisa de revisão pela mesma razão que o lápis precisa de borracha: *Errar é humano*. A segunda razão para precisarmos de revisões técnicas é que embora as pessoas sejam boas para “descobrir” alguns de seus próprios erros, vários tipos de erros escapam mais facilmente daquele que os cometeu do que de outras pessoas externas. O processo de revisão é, portanto, a resposta para a oração de Robert Burns:

Oh! que uma força conceda poder
para nos vermos como os outros nos veem

Uma revisão — qualquer revisão — é uma forma de usar a diversidade de um grupo de pessoas para:

1. Apontar aperfeiçoamentos necessários no produto de uma única pessoa ou de uma equipe.
2. Confirmar aquelas partes de um produto em que aperfeiçoamentos são indesejáveis ou desnecessários.
3. Obter trabalho técnico de qualidade mais uniforme, ou pelo menos mais previsível; qualidade que possa ser alcançada sem revisões, de modo a tornar o trabalho técnico mais gerenciável.

Diversos tipos de revisão podem ser realizados como parte do processo de engenharia de software. Cada um deles tem sua função. Um encontro informal na máquina de café é

PANORAMA

O que é? À medida que desenvolvemos o trabalho de engenharia de software, cometemos erros. Não há nenhum motivo para se envergonhar disso — desde que tentemos, com muita dedicação, encontrar e corrigir os erros antes que sejam passados para os usuários finais. As revisões técnicas são o mecanismo mais efetivo para descobrir erros logo no início da gestão de qualidade.

Quem realiza? Os engenheiros de software realizam as revisões técnicas, também chamadas revisões paritárias, juntamente com seus colegas.

Por que é importante? Ao se descobrir um erro logo no início do processo, fica menos caro corrigi-lo. Além disso, os erros podem aumentar à medida que o processo continua. Portanto, um erro relativamente insignificante, sem tratamento no início do processo, pode ser amplificado e se transformar em um conjunto de erros graves para a sequência do projeto. Finalmente, as revisões minimizam o tempo devido à redução do número

de reformulações que serão necessárias ao longo do projeto.

Quais são as etapas envolvidas? A abordagem em relação às revisões irá variar dependendo do grau de formalidade escolhido. Em geral, são empregadas seis etapas, embora nem todas sejam usadas para todo tipo de revisão: planejamento, preparação, estruturação da reunião, anotação de erros, realização das correções (feita fora da revisão) e verificação se as correções foram feitas apropriadamente.

Qual é o artefato? O artefato de uma revisão é uma lista de problemas e/ou erros que foram descobertos. Além disso, também é indicado o estado técnico do produto resultante.

Como garantir que o trabalho foi realizado corretamente? Primeiramente, escolha o tipo de revisão apropriado para o seu ambiente de desenvolvimento. Siga as diretrizes que levam a revisões bem-sucedidas. Se as revisões realizadas conduzirem a software de maior qualidade, elas foram feitas corretamente.

uma forma de revisão, caso sejam discutidos problemas técnicos. Uma apresentação formal da arquitetura de software para um público formado por clientes, pessoal técnico e gerencial também é uma forma de revisão. Neste livro, entretanto, focalizaremos as *revisões técnicas ou paritárias*, exemplificadas por *revisões informais*, *walkthroughs* e *inspeções*. Uma revisão técnica, RT, é o filtro mais efetivo do ponto de vista de controle da qualidade. Conduzida por engenheiros de software (e outros) para engenheiros de software, a RT é um meio eficaz para revelar erros e aumentar a qualidade do software.

15.1 IMPACTO DE DEFEITOS DE SOFTWARE NOS CUSTOS



Revisões são como um filtro no fluxo de trabalho da gestão da qualidade. Um número muito pequeno de revisões e o fluxo será "sujo". Um número muito grande de revisões e o fluxo diminui muito e vira um gotejamento. Use métricas para determinar quais revisões funcionam e as enfatize. Elimine do fluxo as revisões ineficazes para acelerar o processo.

No contexto da gestão de qualidade, os termos *defeito* e *falha* são sinônimos. Ambos implicam um problema de qualidade que é descoberto *depois* de o software ter sido liberado para os usuários finais (ou para uma outra atividade estrutural dentro da gestão de qualidade). Em capítulos anteriores, usamos o termo *erro* para indicar um problema de qualidade que é descoberto por engenheiros de software (ou outros) *antes* de o software ser liberado ao usuário final (ou para outra atividade estrutural dentro da gestão de qualidade).

O principal objetivo das revisões técnicas é encontrar erros durante o processo, de modo a não se tornarem defeitos depois da liberação do software. O benefício evidente das revisões técnicas é a descoberta precoce de erros, de modo que eles não sejam propagados para a próxima etapa na gestão de qualidade.

Diversos estudos e análise sobre o tema indicam que as atividades de projeto introduzem de 50 a 65% de todos os erros (e, em última instância, todos os defeitos), durante a gestão de qualidade. Entretanto, técnicas de revisão demonstraram ser até 75% eficazes [Jon86] na descoberta de falhas de projeto. Detectando e eliminando um grande percentual desses erros, o processo de revisão reduz substancialmente o custo das atividades subsequentes na gestão de qualidade.

INFORMAÇÕES



Bugs, erros e defeitos

O objetivo do controle da qualidade de software e da gestão da qualidade em geral é, em sentido mais amplo, eliminar problemas de qualidade no software. Tais problemas são conhecidos por diversos nomes — *bugs*, falhas, erros ou defeitos, apenas para citar alguns. Esses termos são sinônimos ou existem diferenças sutis entre eles?

Neste livro é feita uma distinção clara entre erro (um problema de qualidade encontrado antes de o software ser liberado aos usuários finais) e *defeito* (um problema de qualidade encontrado apenas *depois* de o software ter sido liberado aos usuários finais¹). Essa distinção é feita porque os erros e os defeitos podem acarretar impactos econômicos, comerciais, psicológicos e humanos muito diferentes. Os engenheiros de software têm a missão de encontrar e corrigir o maior número possível de erros antes dos clientes e/ou usuários finais. Devem-se evitar defeitos — pois (de modo justificável) criam uma imagem negativa do pessoal de software.

É importante notar, entretanto, que a distinção temporal entre erros e defeitos feita neste livro não é um pensamento dominante. O consenso geral na comunidade de engenharia de software é que defeitos e erros, falhas e *bugs* são sinônimos. Ou seja, o momento em que o problema foi encontrado não tem nenhuma influência no termo usado para descrevê-lo. Parte do argumento a favor desta visão é que, muitas vezes, fica difícil fazer uma distinção clara entre pré e pós-entrega (consideremos, por exemplo, um processo incremental usado no desenvolvimento ágil).

Independentemente da maneira escolhida para interpretar esses termos ou do momento em que um problema é descoberto, o que importa efetivamente é que os engenheiros de software devem se esforçar — *muitíssimo* — para encontrar problemas antes que seus clientes e usuários finais os façam. Caso tenha maior interesse nessa questão, uma discussão razoavelmente completa sobre a terminologia envolvendo *bugs* pode ser encontrada em www.softwaredevelopment.ca/bugs.shtml.

1 Se considerarmos o aperfeiçoamento na gestão de qualidade, um problema de qualidade que se propaga de uma atividade estrutural do processo (por exemplo, **modelagem**) para outra (por exemplo, **construção**) também pode ser chamado de “defeito”, pois o problema deveria ter sido descoberto antes de um produto resultante (por exemplo, um modelo de projeto) ter sido “liberado” para a atividade seguinte.

15.2 AMPLIFICAÇÃO E ELIMINAÇÃO DE DEFEITOS

“Algumas enfermidades, como dizem os médicos, são fáceis de curar logo no início, mas difíceis de diagnosticar... Com o tempo, quando elas não foram diagnosticadas e tratadas no início, tornam-se de fácil diagnóstico, mas difícil cura.”

Nicolau Maquiavel

Um modelo de amplificação de *defeitos* [IBM81] pode ser usado para ilustrar a geração e a detecção de erros durante o projeto e nas ações para geração de código de uma gestão de qualidade. O modelo é ilustrado esquematicamente na Figura 15.1. Um retângulo representa uma ação de engenharia de software. Durante a ação, os erros podem ser gerados inadvertidamente. Pode ser que a revisão falhe na descoberta de erros recentes e erros de etapas anteriores, resultando em uma série de erros que são passados para a etapa seguinte. Em alguns casos, esses erros que são passados e provenientes de etapas anteriores são amplificados (fator de amplificação x) pelo trabalho atual. As subdivisões dos retângulos representam cada uma dessas características e o percentual de eficiência para a detecção de erros, uma função da meticulosidade da revisão.

A Figura 15.2 ilustra um exemplo hipotético da amplificação de defeitos para uma gestão de qualidade em que não foi realizada nenhuma revisão. De acordo com a figura, supõe-se que cada etapa de teste revele e corrija 50% de todos os erros de entrada sem introduzir nenhum erro novo (uma hipótese otimista). Dez defeitos preliminares de projeto são amplificados para 94 erros antes do início do teste. Vinte erros (defeitos) latentes são liberados para o campo. A Figura 15.3 considera as mesmas condições, exceto que as revisões de projeto e código são realizadas como parte de cada ação de engenharia de software. Nesse caso, dez erros de projeto preliminares (de arquitetura) iniciais são amplificados para 24 erros antes do início dos testes. Existem apenas três erros latentes. Podem ser estabelecidos os custos relativos associados à descoberta e à correção dos erros, bem como o custo total (com e sem revisão, para nosso exemplo hipotético). O número de erros revelados durante cada uma das etapas indicadas nas Figuras 15.2 e 15.3 é multiplicado pelo custo para eliminar um erro (1,5 unidades de custo para projeto, 6,5 unidades de custo antes do teste, 15 unidades de custo durante o teste e 67 uni-

FIGURA 15.1

Modelo de amplificação de defeitos

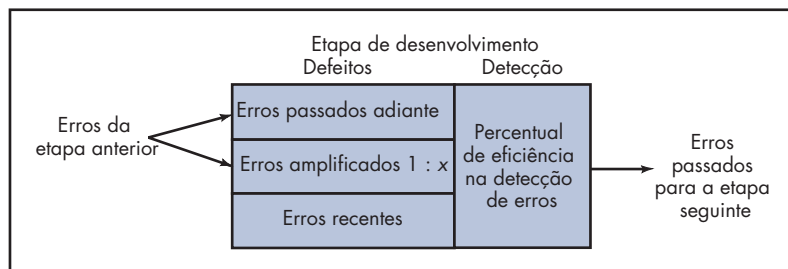


FIGURA 15.2

Amplificação de defeitos – sem revisão

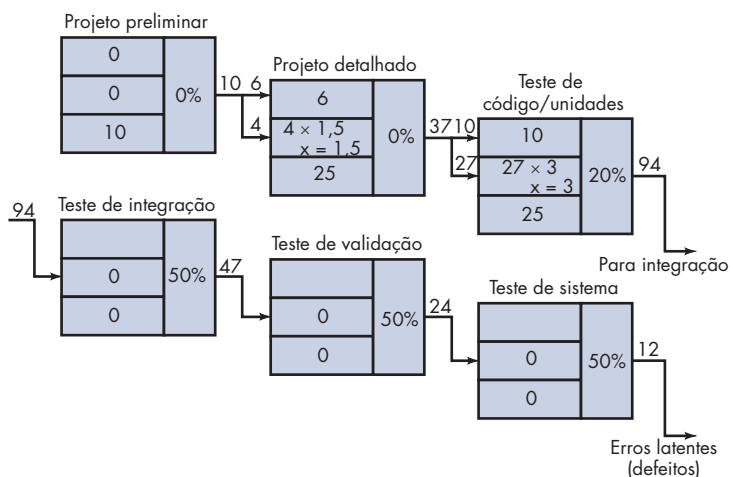
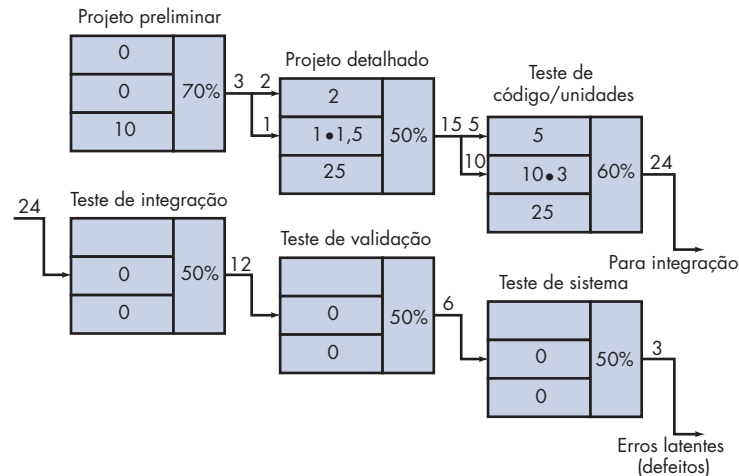


FIGURA 15.3

**Amplificação
de defeitos –
revisões
realizadas**



dades de custo depois da entrega).² Usando esses dados, o custo total para desenvolvimento e manutenção quando são realizadas revisões é de 783 unidades de custo. Quando não é feita nenhuma revisão, o custo total é de 2.177 unidades — aproximadamente três vezes mais caro.

Para realizarmos revisões, temos de despendar tempo e esforço, e a empresa de desenvolvimento tem de disponibilizar recursos financeiros. Entretanto, os resultados do exemplo anterior não deixam nenhuma dúvida de que podemos pagar agora ou então pagar muito mais no futuro.

15.3 MÉTRICAS DE REVISÃO E SEU EMPREGO

As revisões técnicas são algumas das muitas ações exigidas como parte de uma prática de engenharia de software adequada. Cada ação requer dedicação de nossa parte. Já que o esforço disponível para o projeto é finito, é importante para uma organização de engenharia de software compreender a eficácia de cada ação definindo um conjunto de métricas (Capítulo 23) que podem ser usadas para avaliar sua eficácia.

Embora possam ser definidas muitas métricas para as revisões técnicas, um subconjunto relativamente pequeno pode nos dar uma boa ideia da situação. As métricas a seguir podem ser reunidas para cada revisão a ser realizada:

- *Esforço de preparação, Ep* — o esforço (em homens/hora) exigido para revisar um produto resultante antes da reunião de revisão.
- *Esforço de avaliação, Ea* — o esforço (em homens/hora) que é despendido durante a revisão em si.
- *Reformulação esforço, Re* — o esforço (em homens/hora) dedicado à correção dos erros revelados durante a revisão.
- *Tamanho do artefato de software, TPS* — uma medida do tamanho do artefato de software que foi revisto (por exemplo, o número de modelos UML ou o número de páginas de documento ou então o número de linhas de código).
- *Erros secundários encontrados, Errsec* — o número de erros encontrados que podem ser classificados como secundários (exigindo menos para ser corrigidos do que algum esforço pré-especificado).

² Esses multiplicadores são ligeiramente diferentes dos dados apresentados na Figura 14.2, que são mais comuns. Entretanto, servem para ilustrar bem a amplificação dos custos de defeitos.

- *Erros graves encontrados, Err_{graves}* — o número de erros encontrados que podem ser classificados como graves (exigindo mais para ser corrigidos do que algum esforço pré-especificado).

Essas métricas podem ser ainda mais refinadas associando-se o tipo de artefato de software que foi revisto para as métricas.

15.3.1 Análise de métricas

Antes de a análise poder ser iniciada, devem ser realizados alguns cálculos simples. O esforço total de revisão e o número total de erros descobertos são definidos como:

$$E_{\text{revisão}} = E_p + E_a + R_c$$
$$\text{Err}_{\text{tot}} = \text{Err}_{\text{sec}} + \text{Err}_{\text{graves}}$$

A *densidade de erros* representa os erros encontrados por unidade do artefato de software revisto.

$$\text{Densidade de erros} = \frac{\text{Err}_{\text{tot}}}{\text{TPS}}$$

Por exemplo, se um modelo de requisitos for revisado para revelar erros, inconsistências e omissões, seria possível calcular a densidade de erros de várias formas diferentes. O modelo de requisitos contém 18 diagramas UML como parte de um total de 32 páginas de material descritivo. A revisão revela 18 erros secundários e 4 erros graves. Consequentemente, $\text{Err}_{\text{tot}} = 22$. A densidade de erros é de 1,2 erros por diagrama UML ou 0,68 erros por página de modelo de requisitos.

Se as revisões forem realizadas para uma série de tipos de artefatos resultantes diferentes (por exemplo, modelo de necessidades, modelo de projeto, código e casos de teste), a porcentagem de erros revelados para cada revisão pode ser calculada em relação ao número total de erros encontrados para todas as revisões. Além disso, podemos calcular também a densidade de erros para cada artefato.

Assim que forem coletados dados de várias revisões realizadas durante vários projetos, os valores médios da densidade de erros permitem que estimemos o número de erros a ser encontrado em um novo documento, mesmo que não seja revisado. Se, por exemplo, a densidade média de erros para um modelo de requisitos for 0,6 erros por página e um novo modelo de requisitos tiver 32 páginas, uma estimativa grosseira sugeriria que a equipe de software irá encontrar aproximadamente 19 ou 20 erros durante a revisão do documento. Se encontrarmos apenas 6 erros, fizemos um trabalho extremamente bom no desenvolvimento do modelo de requisitos ou então a abordagem de revisão não foi suficientemente completa.

Assim que forem realizados os testes (Capítulos 17 a 20), é possível reunir dados de erros adicionais, incluindo o esforço necessário para encontrar e corrigir erros revelados durante os testes e a densidade de erros do software. Os custos associados à descoberta e à correção de um erro durante um teste podem ser comparados com aqueles obtidos nas revisões. Isso é discutido na Seção 15.3.2.

15.3.2 Eficácia dos custos de revisões

É difícil medir, em tempo real, a eficácia dos custos de qualquer revisão técnica. Uma organização de engenharia de software pode avaliar a eficácia das revisões e seu custo-benefício apenas após terem sido completadas as revisões, reunidas as métricas de revisão e calculados os dados médios. E finalmente, a partir desse ponto, medir a qualidade do software (por meio de testes).

Voltando ao exemplo apresentado na Seção 15.3.1, determinou-se que a densidade média de erros para os modelos de requisitos foi de 0,6 erros por página. Verificou-se que o esforço necessário para corrigir um erro secundário do modelo (imediatamente após a revisão) exige 4 homens/hora. Verificou-se que o esforço necessário para corrigir um erro grave era de 18 homens/hora.

Examinando-se os dados coletados na revisão, determina-se que os erros secundários ocorrem com uma frequência aproximada 6 vezes maior que a de erros graves. Consequentemente, podemos estimar que o esforço médio para encontrar e corrigir um erro de requisitos durante uma revisão é de cerca de 6 homens/hora.

Os erros relacionados aos requisitos, revelados durante os testes, exigem uma média de 45 homens/hora para ser encontrados e corrigidos (não há dados disponíveis sobre a gravidade relativa do erro). Usando as médias observadas, obtemos:

$$\text{Esforço poupado por erro} = \text{Err}_{\text{testes}} - \text{Err}_{\text{revisões}} \\ 45 - 6 = 30 \text{ homens/hora/erro}$$

Como foram encontrados 22 erros durante a revisão do modelo de requisitos, obteríamos uma economia aproximada de 660 homens/hora de esforço de testes. E essa economia refere-se apenas aos erros relacionados ao modelo de requisitos. Os erros associados a projeto e codificação aumentariam ainda mais o benefício geral. Em suma — o esforço poupado nos leva a ciclos de entrega mais curtos e a menor tempo de colocação do produto no mercado.

Em seu livro sobre revisões paritárias, Karl Wiegers [Wie02] discute dados incidentais obtidos de grandes empresas que usaram *inspeções* (um tipo de revisão técnica relativamente formal) como parte de suas atividades de controle da qualidade de software. A Hewlett Packard relatou um retorno sobre o investimento de 10:1 para inspeções e citou que a entrega real do produto foi acelerada, em média, 1,8 mês. A AT&T indicou que as inspeções reduziram o custo total de erros de software por um fator de 10, a qualidade foi incrementada em um grau de magnitude e a produtividade aumentou 14%. Outros relatos informam benefícios semelhantes. As revisões técnicas (para projeto e outras atividades técnicas) fornecem uma relação custo-benefício demonstrável e efetivamente economizam tempo.

Porém, para muitas pessoas da área de software, essa afirmação é contraintuitiva. “As revisões tomam tempo”, argumentam os profissionais de software “não temos muito tempo!”. Eles defendem que tempo é um bem precioso em qualquer projeto de software e a habilidade de revisar “detalhadamente todos os artefatos resultantes” absorve muito tempo.

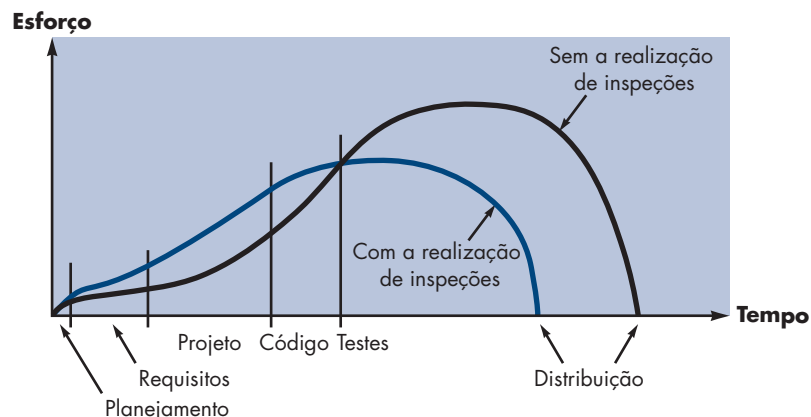
Os exemplos apresentados anteriormente nesta seção indicam o contrário. Mais importante ainda, foram coletados dados do setor para revisões de software por mais de duas décadas e estão sintetizados de forma qualitativa nos gráficos da Figura 15.4.

Em relação à figura, o esforço despendido quando revisões são empregadas aumenta efetivamente no início do desenvolvimento de um módulo de software, mas esse investimento inicial para revisões rende dividendos, pois o esforço de testes e correções é reduzido. Igualmente importante, a data de entrega/distribuição para o desenvolvimento com revisões é anterior àquela sem o uso de revisões. As revisões não gastam tempo, elas poupam!

FIGURA 15.4

Esforço despendido com e sem o emprego de revisões

Fonte: Adaptado de (Fag86)



15.4 REVISÕES: UM ESPECTRO DE FORMALIDADE

As revisões técnicas devem ser aplicadas com um nível de formalidade apropriado para o produto a ser construído, a cronologia do projeto e as pessoas que estão realizando o trabalho. A Figura 15.5 representa um modelo de referência para revisões técnicas [Lai02] que identifica quatro características que contribuem para a formalidade na qual uma revisão é conduzida.

Cada uma das características do modelo de referência ajuda a definir o nível da formalidade da revisão. A formalidade de uma revisão aumenta quando: (1) são definidos explicitamente os papéis distintos para os revisores, (2) há um nível suficiente de planejamento e preparação para a revisão, (3) é definida uma estrutura distinta para a revisão (incluindo tarefas e artefatos internos) e (4) ocorre o follow-up pelos revisores para qualquer correção realizada.

Para compreender o modelo de referência, suponhamos que tenhamos decidido revisar o projeto da interface para o **CasaSeguraGarantida.com**. Isso pode ser feito de várias formas diferentes, que vão desde relativamente informal a extremamente rigorosa. Caso ache que a abordagem informal é a mais apropriada, solicite a alguns colegas (pares) para examinarem o protótipo da interface em uma tentativa de descobrir problemas potenciais. Todos os participantes decidem que não haverá nenhuma preparação prévia, mas que você irá avaliar o protótipo de uma forma razoavelmente estruturada — verificando primeiramente o layout, em seguida a estética, depois as opções de navegação e assim por diante. Na qualidade de projetista, você decide fazer algumas anotações, mas nada formal.

Mas o que acontece se a interface for fundamental para o sucesso do projeto inteiro? E se vidas humanas dependerem de uma interface ergonomicamente consistente? Será necessária uma abordagem mais rigorosa. Deverá ser formada uma equipe de revisão. Cada membro da equipe deve ter um papel específico a ser desempenhado — liderar a equipe, registrar as descobertas, apresentar o material e assim por diante. Cada revisor deve ter acesso ao artefato de software (neste caso, o protótipo da interface) antes da revisão e gastar tempo procurando erros, inconsistências e omissões. Um conjunto de tarefas específicas necessita ser conduzido, tomando como base uma agenda desenvolvida antes de a revisão ocorrer. Os resultados da revisão precisam ser registrados formalmente, e a equipe deve decidir sobre a situação do artefato de software com base na revisão. Os membros da equipe de revisão também podem verificar se as correções foram feitas de forma apropriada.

Neste livro, consideramos duas grandes categorias de revisões técnicas: revisões informais e revisões técnicas mais formais. Dentro de cada categoria, podem ser escolhidas várias abordagens distintas que são apresentadas nas seções seguintes.

FIGURA 15.5

**Modelo de referência
para revisões
técnicas**



15.5 REVISÕES INFORMAIS

Entre as revisões informais temos um simples teste de mesa de um artefato de engenharia de software (com um colega), uma reunião informal (envolvendo mais de duas pessoas) com a finalidade de revisar um artefato, ou os aspectos orientados a revisões da programação em pares (Capítulo 3).

Um *simples teste de mesa* ou uma *reunião informal* realizada com um colega é uma revisão. Entretanto, pelo fato de não haver nenhum planejamento ou preparação antecipados, nenhuma agenda ou estrutura de reuniões e nenhum follow-up sobre os erros encontrados, a eficácia de tais revisões é consideravelmente menor que as abordagens mais formais. Mas um simples teste de mesa pode realmente revelar erros que, de outra forma, poderiam se propagar ainda mais na gestão de qualidade.

Uma forma de aumentar a eficácia de uma revisão do tipo teste de mesa é desenvolver um conjunto de listas de verificação simples para cada artefato produzido pela equipe de software. As questões levantadas na lista de verificação são genéricas, mas servirão como guia para os revisores verificarem o produto resultante. Por exemplo, vamos reexaminar um teste de mesa do protótipo da interface para **CasaSeguraGarantida.com**. Em vez de simplesmente ficar testando o protótipo na estação de trabalho do projetista, o projetista e um colega examinam o protótipo usando uma lista de verificação para interfaces:

O layout é projetado usando convenções padronizadas? Da esquerda para a direita? De cima para baixo?

- A apresentação precisa de barra de rolagem?
- A cor e o posicionamento, o tipo e o tamanho dos elementos são usados efetivamente?
- Todas as opções de navegação ou funções representadas estão no mesmo nível de abstração?
- Todas as opções de navegação são claramente identificadas?

e assim por diante. Quaisquer erros ou problemas verificados pelos revisores são registrados pelo projetista para resolução mais tarde. Poderiam ser programados testes de mesa de forma *ad hoc* ou eles seriam compulsórios, como parte da boa prática de engenharia de software. Em geral, a quantidade de material a ser revisada é relativamente pequena e o tempo total gasto em um teste de mesa vai um pouco além de uma ou duas horas.

No Capítulo 3, descrevemos a *programação em pares* da seguinte maneira: “XP (eXtreme Programming) recomenda que duas pessoas trabalhem juntas em uma mesma estação de trabalho para criar código. Isso disponibiliza um mecanismo para a resolução de problemas em tempo real (duas cabeças normalmente funcionam melhor do que uma) e a garantia da qualidade em tempo real”.

A programação em pares pode ser caracterizada como um teste de mesa contínuo. Em vez de programar uma revisão em algum momento, a programação em pares encoraja a revisão contínua enquanto um artefato de software (projeto ou código) é criado. O benefício é a descoberta imediata de erros e, conseqüentemente, maior qualidade do artefato final.

Em sua discussão sobre a eficácia da programação em pares, Williams e Kessler [Wil00] afirmam:

Tanto evidências incidentais quanto evidências estatísticas iniciais indicam que a programação em pares é uma técnica poderosa para gerar, de forma produtiva, produtos de software de alta qualidade. O par trabalha e compartilha ideias juntos para abordar as complexidades do desenvolvimento de software. Eles realizam inspeções continuamente em cada um dos artefatos do outro, levando à forma mais precoce e eficiente possível de eliminação de defeitos. Além disso, cada um faz o outro se manter atentamente focado na tarefa em questão.

Alguns engenheiros de software sustentam que a redundância inerente da programação em pares é um desperdício de recursos. Afinal de contas, por que alocar duas pessoas para um

trabalho que uma única é capaz de realizar? A resposta a essa pergunta pode ser encontrada na Seção 15.3.2. Se a qualidade do produto resultante da programação em pares for significativamente melhor que o trabalho individual, as economias relacionadas com qualidade são plenamente capazes de justificar a “redundância” implícita na programação em pares.



Listas de verificação para revisões

Mesmo quando as revisões são bem organizadas e apropriadamente conduzidas, não é uma má ideia munir os revisores de um recurso a mais. Vale a pena ter uma lista de verificação que forneça a cada revisor as perguntas que deveriam ser feitas sobre o artefato específico que está passando por revisão.

Um dos conjuntos mais completos de listas de verificação para revisões foi desenvolvido pela NASA no Goddard Space Flight Center e se encontra disponível em <http://sw-assurance.gsfc.nasa.gov/disciplines/quality/index.php>

Outras listas de verificação para revisões técnicas muito úteis também foram propostas por:

Process Impact (www.processimpact.com/pr_goo_dies.shtml)

Software Dioxide (www.softwaredioxide.com/channels/con.View.asp?id=6309)

Macadamian (www.macadamian.com)

The Open Group Architecture Review Checklist (www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm)

DFAS [downloadable] (www.dfas.mil/technology/pal/ssps/docstds/spm036.doc)

INFORMAÇÕES

15.6 REVISÕES TÉCNICAS FORMAIS

“Não há impulso maior do que aquele de um homem revisar o trabalho de outro homem.”

Mark Twain

Revisão técnica formal ou *RTF* é uma atividade de controle da qualidade de software realizada por engenheiros de software (e outros profissionais). Os objetivos de uma RTF são: (1) descobrir erros na função, lógica ou implementação para qualquer representação do software; (2) verificar se o software que está sendo revisado atende aos requisitos; (3) garantir que o software foi representado de acordo com padrões predefinidos; (4) obter software que seja desenvolvido de maneira uniforme; e (5) tornar os projetos mais gerenciáveis. Além disso, a RTF serve como base de treinamento, possibilitando que engenheiros mais novos observem diferentes abordagens para análise, projeto e implementação de software.

A RTF também serve para promover backup e continuidade, pois muitas pessoas se familiarizam com partes do software que de outra maneira jamais teriam visto.

A RTF é, atualmente, uma classe de revisões que inclui *walkthroughs* e *inspeções*. Cada RTF é realizada como uma reunião e apenas será bem-sucedida se for apropriadamente planejada, controlada e tiver a participação de todos os envolvidos. Nas seções a seguir são apresentadas orientações similares às aquelas para um walkthrough na forma de uma revisão técnica formal representativa. Caso tenha interesse em inspeções de software, bem como queira informações adicionais sobre walkthroughs, ver [Rad02], [Wie02] ou [Fre90].

15.6.1 Uma reunião de revisão

Independentemente do formato de RTF escolhido, cada reunião de revisão deve observar as seguintes restrições:

- Devem estar envolvidas de três a cinco pessoas em uma revisão (tipicamente).
- Deve ocorrer uma preparação antecipada, porém não deve tomar mais do que duas horas de trabalho de cada pessoa.
- A duração da reunião de revisão deve ser de menos de duas horas.

Dadas essas restrições, deve ser óbvio que uma RTF se concentre em uma parte específica (e pequena) do software. Por exemplo, em vez de tentar revisar um projeto inteiro, os walkthroughs são realizados para cada componente ou para um pequeno grupo de componentes. Afunilando-se o foco, a RTF terá maior probabilidade de revelar erros.

WebRef

O NASA SATC *Formal Inspection Guidebook* pode ser copiado de satc.gsfc.nasa.gov/Documents/fi/gdb/fi.pdf.

PONTO-CHAVE

Uma RTF concentra-se em uma parte relativamente pequena de um artefato.



Em algumas situações, é uma boa ideia fazer com que outra pessoa que não seja o produtor analise o produto que está sendo submetido a uma revisão. Isso leva a uma interpretação literal do artefato e a um melhor diagnóstico dos erros.

O foco da RTF é um artefato resultante (por exemplo, parte de um modelo de requisitos, o projeto detalhado de um componente, o código-fonte de um componente). O indivíduo que desenvolveu o artefato — o *produtor* — informa ao líder de projeto do artefato está completo e que é necessário fazer uma revisão. O líder de projeto contata um *líder de revisão*, que avalia o artefato em termos de completude, gera cópias dos materiais resultantes e as distribui para dois ou três *revisores* para preparação prévia. Espera-se que cada revisor gaste de uma a duas horas revisando o artefato, tomando notas e, de alguma forma, se familiarizando com o trabalho realizado. Ao mesmo tempo, o líder da reunião de revisão também revisa o artefato e estabelece uma agenda para a reunião de revisão, que normalmente é marcada para o dia seguinte.

Uma reunião de revisão tem a participação de um líder de revisão, todos os revisores e o produtor. Um dos revisores assume o papel de *registrador*; isto é, o indivíduo que registra (por escrito) todas as questões importantes surgidas durante a revisão. A RTF começa com uma introdução da agenda e uma breve introdução por parte do produtor, que continua na descrição do artefato resultante, explicando o material, enquanto os revisores levantam questões com base em sua preparação prévia. Quando são descobertos problemas ou erros válidos, o registrador toma nota de cada um deles.

No final da revisão, todos os participantes da RTF devem decidir se: (1) aceitam o artefato sem as modificações adicionais, (2) rejeitam o artefato devido a erros graves (uma vez corrigidos, deve ser realizada outra revisão) ou (3) aceitam o artefato provisoriamente (foram encontrados erros secundários que devem ser corrigidos, mas não haverá nenhuma outra revisão). Após uma tomada de decisão, todos os participantes da RTF assinam um documento de aprovação, indicando sua participação na revisão e sua concordância com as descobertas da equipe de revisão.

15.6.2 Relatório de revisão e manutenção de registros

Durante a RTF, um revisor (o registrador) registra ativamente todos os problemas levantados. Estes são sintetizados no final da reunião de revisão e é produzida uma *lista de problemas de revisão*. Além disso, um *relatório sintetizado da revisão técnica formal* é completado. O relatório sintetizado de revisão deve responder a três questões:

1. O que foi revisado?
2. Quem o revisou?
3. Quais foram as descobertas e as conclusões?

Um relatório sintetizado da revisão é um formulário de uma página (com possíveis anexos). Ele torna-se um registro histórico do projeto e pode ser distribuído ao líder do projeto e a outras partes interessadas.

A lista de problemas de revisão atende a dois propósitos: (1) identificar áreas problemáticas no artefato e (2) servir como uma lista de verificação de itens de ação que orienta o produtor à medida que são feitas as correções. Normalmente é anexada uma lista de problemas ao relatório sintetizado.

Devemos estabelecer um procedimento de acompanhamento para garantir que itens contidos na lista de problemas tenham sido corrigidos apropriadamente. Caso isso não seja feito, é possível que problemas levantados possam “ficar para trás”. Uma das abordagens é atribuir a responsabilidade pelo acompanhamento (follow-up) ao líder da revisão.

15.6.3 Diretrizes de revisão

Devem-se estabelecer previamente diretrizes para a realização de revisões técnicas formais, distribuídas a todos os revisores, ter a anuência de todos e, então, segui-las à risca. Uma revisão não controlada muitas vezes pode ser pior do que não fazer nenhuma revisão. A seguir, apresentamos um conjunto mínimo de diretrizes para revisões técnicas formais:



Não aponte erros de forma áspera. Uma maneira de ser gentil é fazer uma pergunta que possibilite ao produtor descobrir o próprio erro.

“Uma reunião é, muitas vezes, um evento em que minutos são perdidos e horas são desperdiçadas.”

Autor desconhecido

“É uma das mais belas recompensas da vida, que nenhum homem pode, sinceramente, tentar ajudar outro sem ajudar a si mesmo.”

Ralph Waldo Emerson

1. *Revisar o produto, não o produtor.* Uma RTF envolve pessoas e egos. Conduzida de forma apropriada, a RTF deve deixar todos os seus participantes com uma agradável sensação de dever cumprido. Conduzido de forma imprópria, a RTF pode assumir a aura de uma inquisição. Os erros devem ser apontados gentilmente; o clima da reunião deve ser descontraído e construtivo; o intuito não deve ser o de causar embaraços ou menosprezo. O líder da revisão deve conduzir a reunião de revisão de tal forma a garantir que o clima seja mantido e as atitudes sejam apropriadas; além disso, deve interromper imediatamente uma revisão que começou a sair do controle.
2. *Estabelecer uma agenda e mantê-la.* Um dos principais males de reuniões de todos os tipos é desviar do foco. Uma RTF deve ser mantida em seu rumo e prazo estabelecidos. O líder da revisão tem a responsabilidade de manter o cronograma da reunião e não deverá ficar receoso em alertar as pessoas quando ela estiver saindo do foco.
3. *Limitar debates e refutação.* Quando uma questão é levantada por um revisor, talvez não haja um acordo universal sobre seu impacto. Em vez de perder tempo debatendo a questão, o problema deve ser registrado para posterior discussão, fora da reunião.
4. *Enunciar as áreas do problema, mas não tentar resolver todo problema registrado.* Uma revisão não é uma sessão para resolução de problemas. A solução de um problema pode, muitas vezes, ser realizada pelo próprio produtor ou com a ajuda de apenas outro colega. A resolução de problemas deve ser adiada para depois da reunião de revisão.
4. *Enunciar as áreas do problema mas não tentar resolver todo o problema registrado.* Uma visão não é uma sessão para resolução de problemas. A solução de um problema pode, muitas vezes, ser realizada pelo próprio produtor ou com a ajuda de apenas outro colega. A resolução de problemas deve ser adiada para depois da reunião de revisão.
5. *Tomar notas.* Algumas vezes é uma boa ideia para o registrador fazer apontamentos em um quadro, de modo que os termos e as prioridades possam ser avaliados por outros revisores à medida que as informações são registradas. Alternativamente, as anotações podem ser introduzidas diretamente em um notebook.
6. *Limitar o número de participantes e insistir na preparação antecipada.* Duas cabeças funcionam melhor do que uma, mas catorze cabeças não funcionam, necessariamente, melhor do que quatro. Mantenha o número de pessoas envolvidas no mínimo necessário. Entretanto, todos os membros da equipe de revisão devem se preparar com antecedência. O líder da revisão deve solicitar comentários escritos (fornecendo uma indicação de que o revisor reviu o material).
7. *Desenvolver uma lista de verificação para cada artefato que provavelmente será revisado.* A lista de verificação ajuda o líder da revisão a estruturar a RTF e auxilia cada revisor a se concentrar nas questões importantes. As listas de verificação devem ser desenvolvidas para análise, projeto, código e até mesmo para o teste dos artefatos.
8. *Alocar os recursos e programar o tempo para as RTFs.* Para as revisões serem eficazes, elas devem ser programadas como tarefas durante a gestão de qualidade. Além disso, deve-se programar o tempo para as inevitáveis modificações que ocorrerão como resultado de uma RTF.
9. *Realizar treinamento significativo para todos os revisores.* Para serem eficazes, todos os participantes de uma revisão deveriam receber algum treinamento formal. O treinamento deve enfatizar tanto questões relacionadas a processos, como o lado psicológico das revisões. Freedman e Weinberg [Fre90] estimam uma curva de aprendizado para cada vinte pessoas que participarão efetivamente de revisões.
10. *Revisar revisões iniciais.* Um interrogatório pode ser benéfico na descoberta de problemas com o próprio processo de revisão. Os primeiros artefatos a ser revisados devem ser as próprias diretrizes de revisão.

Como muitas variáveis (por exemplo, o número de participantes, o tipo de artefatos resultantes, o timing, ou tempo adequado e a duração, a abordagem de revisão específica) causam im-

pacto sobre uma revisão bem-sucedida, uma organização de software deve fazer experimentos para determinar qual abordagem funcionará melhor em um contexto local.

15.6.4 Revisões por amostragem

Em um ambiente ideal, todo artefato de engenharia de software deveria passar por uma revisão técnica formal. No mundo real dos projetos de software, os recursos são limitados e o tempo é escasso. Como consequência, as revisões são muitas vezes esquecidas, muito embora seu valor como um mecanismo de controle de qualidade seja reconhecido.

Thelin e seus colegas [The01] sugerem um processo de revisão por amostragem em que amostras de todos os artefatos da engenharia de software sejam inspecionadas para determinar quais são mais suscetíveis a erro. Recursos completos de RTF são, então, direcionados apenas para os artefatos com maior probabilidade de ser suscetíveis a erro (com base em dados coletados durante a amostragem).

Para ser eficaz, o processo de revisão por amostragem deve tentar quantificar aqueles produtos de trabalho que são alvos primários para as RTFs completas. Para conseguir isso, são sugeridas as seguintes etapas [The01]:

1. Inspeccionar uma fração a_i de cada artefato de software resultante i . Registrar o número de falhas f_i encontradas em a_i .
2. Desenvolver uma estimativa total do número de falhas contido no artefato i multiplicando f_i por $1/a_i$.
3. Classificar os artefatos em ordem decrescente, de acordo com a estimativa total do número de falhas contidas em cada um deles.
4. Concentrar recursos de revisão disponíveis naqueles artefatos que possuem o maior número estimado de falhas.



As revisões tomam tempo, mas é um tempo bem empregado. Entretanto, se o tempo for reduzido e você não tiver nenhuma outra opção, não descarte as revisões. Em vez disso, use revisões por amostragem.

CASA SEGURA



Problemas de qualidade

Cena: Sala de Doug Miller quando se inicia o projeto de software CasaSegura.

Atores: Doug Miller (gerente da equipe de engenharia de software do CasaSegura e outros membros da equipe de engenharia de software do produto).

Conversa:

Doug: Sei que não investimos tempo para desenvolver um plano de qualidade para este projeto, mas nós já estamos nele e temos de considerar a qualidade... Certo?

Jamie: Certamente. Já decidimos que enquanto desenvolvermos o modelo de requisitos [Capítulos 6 e 7], Ed se comprometeu a desenvolver um procedimento de testes para cada requisito.

Doug: Isso é realmente interessante, mas não iremos esperar até que os testes avaliem a qualidade, não é mesmo?

Vinod: Não! Obviamente, não. Temos revisões programadas no plano de projeto para este incremento de software. Começaremos o controle da qualidade com as revisões.

Jamie: Estou bastante preocupado, pois acredito que não teremos tempo suficiente para realizar todas as revisões. Na realidade, eu sei que não teremos.

Doug: Huumm. Então o que você sugere?

Jamie: Acho que devemos escolher aqueles elementos mais críticos dos modelos de requisitos e de projeto e os revisarmos.

Vinod: Mas e se deixarmos alguma coisa de lado em uma parte do modelo em que não fizemos uma revisão?

Shakira: Li alguma coisa sobre uma técnica de amostragem [Seção 15.6.4] que poderia nos ajudar a determinar candidatos a uma revisão. (Shakira explica a abordagem.)

Jamie: Talvez... Mas não estou certo se teremos tempo até mesmo para amostrar cada elemento dos modelos.

Vinod: O que você quer que façamos, Doug?

Doug: Usemos algo da *Extreme Programming (Programação Extrema)* [Capítulo 3]. Desenvolveremos os elementos de cada modelo em pares — duas pessoas — e faremos uma revisão informal de cada um deles à medida que prosseguimos. Em seguida, separaremos elementos “críticos” para uma revisão mais formal em equipe, mas manteremos essas revisões em um número mínimo. Dessa forma, tudo será examinado por mais de uma pessoa, mas ainda manteremos nossas datas de entrega.

Jamie: Isso significa que teremos de revisar o cronograma.

Doug: Que assim seja. A qualidade prevalece sobre o cronograma nesse projeto.

A fração do trabalho que é amostrada deve ser representativa do artefato como um todo e suficientemente grande para ser significativa para os revisores que fazem a amostragem. À medida que a_i aumenta, a probabilidade de que a amostra seja uma representação válida do artefato também aumenta. Entretanto, os recursos necessários para realizar a amostragem também aumentam. Uma equipe de engenharia de software deve estabelecer o melhor valor para a_i para os tipos de artefatos produzidos.³

15.7 RESUMO

O intuito de toda revisão técnica é encontrar erros e revelar problemas que teriam um impacto negativo sobre o software a ser entregue. Quanto antes um erro for descoberto e corrigido, menor a probabilidade de que esse erro se propague a outros artefatos de engenharia de software, amplificando o problema e resultando em um esforço significativamente maior para corrigi-lo.

Para determinar se as atividades de controle da qualidade estão funcionando, deve-se reunir um conjunto de métricas. As métricas de revisão concentram-se no esforço exigido para conduzir uma revisão e nos tipos e gravidade dos erros revelados durante a revisão. Assim que os dados sobre métricas tiverem sido coletados, eles poderão ser usados para avaliar a eficácia das revisões realizadas. Os dados do setor indicam que as revisões geram um retorno significativo sobre o investimento.

Um modelo de referência da formalidade de uma revisão identifica os papéis desempenhados pelas pessoas, o planejamento e a preparação, a estrutura das reuniões, a abordagem e a verificação da correção como características que indicam o nível de formalidade em que uma revisão é conduzida. As revisões informais são superficiais por natureza, mas mesmo assim podem ser usadas efetivamente na descoberta de erros. As revisões formais são mais estruturadas e têm maior probabilidade de resultarem em software de alta qualidade.

As revisões informais caracterizam-se por um mínimo de planejamento e de preparação e um pouco de manutenção de registros. Os testes de mesa e a programação em pares caem na categoria de revisão informal.

Uma revisão técnica formal é uma reunião estilizada que se mostrou extremamente eficaz na revelação de erros. Os walkthroughs e as inspeções estabelecem papéis definidos para cada revisor, encorajam a antecipação do planejamento e da preparação, exigem a aplicação de diretrizes de revisão já definidas e tornam compulsórios a manutenção de registros e o relatório de estado das revisões. As revisões por amostragem podem ser usadas quando não é possível realizar revisões técnicas formais para todos os artefatos.

PROBLEMAS E PONTOS A PONDERAR

15.1. Explique a diferença entre *erro* e *defeito*.

15.2. Por que não podemos simplesmente aguardar até que os testes terminem para descobrir e corrigir todos os erros de software?

15.3. Suponha que tenham sido introduzidos 10 erros no modelo de requisitos e que cada erro será amplificado no projeto detalhado por um fator de 2:1 e que outros 20 erros de projeto sejam introduzidos e então amplificados na razão de 1,5:1 no código onde mais 30 erros são introduzidos. Suponha ainda que o teste de todas as unidades irá encontrar 30% de todos os erros, a integração descobrirá 30% dos erros remanescentes e os testes de validação encontrarão 50% dos erros restantes. Não é realizada nenhuma revisão. Quantos erros serão liberados para o campo?

³ Thelin e seus colegas realizaram uma simulação detalhada que pode ajudar a fazer essa determinação. Ver[The01] para mais detalhes.

15.4. Reconsidere a situação descrita no Problema 15.3, mas suponha agora que sejam efetuadas as revisões de requisitos, de projeto e de código e estas terão uma eficiência de 60% na descoberta de todos os erros nessa etapa. Quantos erros chegarão ao campo?

15.5. Reconsidere a situação descrita nos Problemas 15.3 e 15.4. Se cada um dos erros que chegam aos usuários custar US\$ 4.800 para ser encontrado, e corrigido e cada erro encontrado na revisão custar US\$ 240 para ser encontrado e corrigido, quanto é poupado em termos monetários com a realização das revisões?

15.6. Descreva com suas próprias palavras o significado da Figura 15.4.

15.7. Qual das características do modelo de referência você imagina que possua a maior influência sobre a formalidade da revisão? Justifique.

15.8. Você seria capaz de imaginar alguns casos em que um teste de mesa poderia criar problemas em vez de gerar benefícios?

15.9. A revisão técnica formal é eficaz apenas se todos estiverem preparados com antecedência. Como se reconhece um participante da revisão que não se preparou? O que você faria caso fosse o líder da revisão?

15.10. Considerando-se todas as diretrizes para a revisão apresentada na Seção 15.6.3, o que você acha que é mais importante e por quê?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Há um número relativamente pequeno de livros sobre revisões de software. Entre as edições recentes que dão uma orientação útil temos os de Wong (Modern Software Review, IRM Press, 2006), Radice (High Quality, Low Cost Software Inspections, Paradoxicon Publishers, 2002), Wiegers (Peer Reviews in Software: A Practical Guide, Addison-Wesley, 2001) e Gilb e Graham (Software Inspections, Addison-Wesley, 1993). Freedman e Weinberg (Handbook of Walkthroughs, Inspections and Technical Reviews, Dorset House, 1990) permanece um clássico e continua a fornecer informações úteis sobre esse tema importante.

Uma ampla gama de fontes de informação sobre revisões de software se encontra à disposição na Internet. Uma lista atualizada de referências que são relevantes sobre revisões de software pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

GARANTIA DA QUALIDADE DE SOFTWARE

CONCEITOS-CHAVE

abordagens formais.....	392
confiabilidade de software.....	395
metas.....	390
padrão ISO 9001:2000....	398
proteção de software.....	396
Seis Sigma.....	394
SQA	
elementos de...	388
estatística.....	393
plano.....	398
tarefas.....	390

A abordagem de engenharia de software descrita neste livro tem um único objetivo: produzir software de alta qualidade em tempo condizente com as necessidades dos interessados. Mesmo assim, muitos leitores sentir-se-ão desafiados pela questão: “O que é qualidade de software?” Philip Crosby [Cro79], em seu livro histórico sobre qualidade, fornece uma resposta sarcástica a esta pergunta:

O problema da gestão da qualidade não é o que as pessoas não sabem a seu respeito. Mas sim, o que eles pensam que sabem...

Nesse aspecto, a qualidade tem muita semelhança com o sexo. Todo mundo está disposto a fazê-lo (sob certas condições, é claro). Todo mundo tem a impressão de que entende de sexo (muito embora não queiram fornecer explicações). Todo mundo pensa que sua execução é apenas uma questão de seguir instintos naturais (afinal de contas, de alguma forma consegue-se fazê-lo). E, obviamente, a maioria das pessoas acredita que os problemas nessas áreas são causados pelos outros (se ao menos eles se preocupassem em fazer as coisas certas...).

De fato, qualidade é um conceito desafiador — este conceito é apresentado em detalhes no Capítulo 14.¹

Alguns desenvolvedores de software continuam a acreditar que a qualidade de software é algo sobre o qual começamos a nos preocupar depois que o código é gerado. Nada poderia estar tão distante da verdade! A *garantia da qualidade de software*, SQA (*software quality assurance*, muitas vezes denominada *gestão da qualidade*) é uma atividade universal (Capítulo 2) aplicada em toda a gestão de qualidade.

PANORAMA

O que é? Não basta dizer simplesmente que a qualidade de software é importante. É preciso: (1) definir explicitamente o seu significado, o que realmente se quer dizer com “qualidade de software”, (2) criar um conjunto de atividades que ajude a garantir que todo artefato resultante da engenharia de software apresente alta qualidade, (3) realizar atividades de garantia e controle da qualidade de software em todos os projetos de software, (4) usar métricas para desenvolver estratégias para aperfeiçoar a gestão da qualidade e, conseqüentemente, a qualidade do produto final.

Quem realiza? Todos os envolvidos no processo de engenharia de software são responsáveis pela qualidade.

Por que é importante? Pode-se fazer certo da primeira vez ou então fazer tudo de novo. Se uma equipe de software buscar a qualidade em todas as atividades de engenharia de software, a quantidade de reformulações a ser feitas é reduzida. Isso resulta em custos menores e, mais importante, menor tempo para disponibilização do produto no mercado.

Quais são as etapas envolvidas? Antes das atividades de garantia da qualidade de software (SQA, *software quality assurance*) iniciarem, é importante definir *qualidade de software* em diferentes níveis de abstração. A partir do momento em que se entende o que é qualidade, a equipe de software deve identificar um conjunto de atividades de SQA que irão filtrar erros do artefato antes de ser passados adiante.

Qual é o artefato? É criado um Plano de Garantia da Qualidade de Software para definir a estratégia de SQA de uma equipe de software. Durante a modelagem e a codificação, o artefato principal da SQA é o resultado das revisões técnicas (Capítulo 15). Durante os testes (Capítulos 17 a 20), são produzidos procedimentos e planos de testes. Também podem ser gerados outros produtos associados ao aperfeiçoamento do processo.

Como garantir que o trabalho foi realizado corretamente? Encontrar erros antes de se tornarem defeitos! Ou seja, trabalhar para melhorar a eficiência de remoção dos defeitos (Capítulo 23) reduzindo, portanto, a quantidade de reformulações que a equipe de software terá de realizar.

¹ Caso não tenha lido o Capítulo 14, você deve fazê-lo agora.

A garantia da qualidade de software (SQA) engloba: (1) um processo de SQA, (2) tarefas específicas de garantia da qualidade e controle da qualidade (inclusive revisões técnicas e uma estratégia de testes multiescalonados), (3) prática efetiva de engenharia de software (métodos e ferramentas), (4) controle de todos os artefatos de software e as mudanças feitas nesses produtos (Capítulo 22), (5) um procedimento para garantir a conformidade com os padrões de desenvolvimento de software (quando aplicáveis) e (6) mecanismos de medição e de relatórios.

Este capítulo concentra-se em problemas de gerenciamento e em atividades específicas de processos que permitem garantir a uma organização de software fazer “as coisas certas, no momento certo e da maneira certa”.

16.1 PROBLEMAS DE BACKGROUND

A garantia e o controle da qualidade são atividades essenciais para qualquer empresa de produtos a ser usados por terceiros. Antes do século vinte, o controle de qualidade era responsabilidade exclusiva do artesão que construía um produto. À medida que o tempo foi passando e técnicas de produção em massa tornaram-se comuns, o controle de qualidade tornou-se uma atividade realizada por outras pessoas e não aquelas que constroem o produto.

“Você cometeu os piores erros.”

Yogi Berra

A primeira função formal de garantia e controle da qualidade foi introduzida no Bell Labs em 1916 e difundiu-se rapidamente no mundo da manufatura. Durante a década de 1940, foram sugeridas abordagens mais formais para o controle de qualidade, fundamentadas em medições e aperfeiçoamento contínuo dos processos [Dem86] como elementos-chave da gestão da qualidade.

Hoje em dia, toda empresa possui mecanismos para garantir a qualidade de seus produtos. Na realidade, declarações explícitas da preocupação de uma empresa com a qualidade tornaram-se um estratagema de marketing nas últimas décadas.

A história da garantia da qualidade no desenvolvimento de software é análoga à história da qualidade na fabricação de hardware. Durante os primórdios da computação (décadas de 1950 e 1960), a qualidade era responsabilidade exclusiva do programador. Padrões para a garantia da qualidade foram introduzidos no desenvolvimento de software por parte de empresas terceirizadas pela indústria militar durante a década de 1970 e se difundiram rapidamente para o desenvolvimento de software no mundo comercial [IEE93]. Estendendo a definição apresentada anteriormente, a garantia da qualidade de software é um “padrão de ações planejado e sistematizado” [Sch98c], ações essas exigidas para garantir alta qualidade no software. O escopo da responsabilidade da garantia da qualidade poderia ser mais bem caracterizado parafraseando-se um famoso comercial de uma indústria automobilística: “A Qualidade é a Tarefa n.º 1”. A implicação para a área de software é que os elementos distintos têm as suas responsabilidades sobre a garantia da qualidade de software — engenheiros de software, gerentes de projeto, clientes, vendedores e os indivíduos que trabalham em um grupo de SQA.

O grupo de SQA funciona como um serviço de defesa do cliente. Ou seja, as pessoas que realizam a SQA devem examinar o software sob o ponto de vista do cliente. O software atende adequadamente aos fatores de qualidade citados no Capítulo 14? O desenvolvimento de software foi conduzido de acordo com os padrões preestabelecidos? As disciplinas técnicas desempenharam apropriadamente seus papéis como parte da atividade de SQA? O grupo de SQA tenta responder a essas e outras perguntas para garantir que a qualidade de software seja mantida.

16.2 ELEMENTOS DE GARANTIA DA QUALIDADE DE SOFTWARE

A garantia da qualidade de software engloba um amplo espectro de preocupações e atividades que se concentram na gestão da qualidade de software e que podem ser sintetizadas da seguinte maneira [Hor03]:

Padrões. O IEEE, a ISO e outras organizações de padronizações produziram uma ampla gama de padrões para engenharia de software e os respectivos documentos. Os padrões

WebRef

Uma discussão aprofundada da SQA, incluindo uma ampla gama de definições, pode ser obtida em www.swqual.com/newsletter/vol2/no1/vol2no1.html.

podem ser adotados voluntariamente por uma organização de engenharia de software ou impostos pelo cliente ou outros interessados. O papel da SQA é garantir que padrões que tenham sido adotados sejam seguidos e que todos os produtos resultantes estejam em conformidade com eles.

Revisões e auditorias. As revisões técnicas são uma atividade de controle de qualidade realizada por engenheiros de software para engenheiros de software (Capítulo 15). Seu intuito é o de revelar erros. Auditorias são um tipo de revisão efetuado pelo pessoal de SQA com o intuito de assegurar-se de que as diretrizes de qualidade estejam sendo seguidas no trabalho de engenharia de software. Por exemplo, uma auditoria do processo de revisão pode ser realizada para garantir que as revisões estejam sendo realizadas de maneira que conduza à maior probabilidade de descoberta de erros.

Testes. Os testes de software (Capítulos 17 a 20) são uma função de controle de qualidade com um objetivo principal — descobrir erros. O papel da SQA é garantir que os testes sejam planejados apropriadamente e conduzidos eficientemente de modo que se tenha a maior probabilidade possível de alcançar seu objetivo primário.

Coleta e análise de erros/defeitos. A única forma de melhorar é medir o nosso desempenho. A SQA reúne e analisa dados de erros e defeitos para melhor compreender como os erros são introduzidos e quais atividades de engenharia de software melhor se adequam para sua eliminação.

Gerenciamento de mudanças. As mudanças são um dos aspectos mais negativos de qualquer projeto de software. Se não forem administradas apropriadamente, podem gerar confusão, e confusão quase sempre leva a uma qualidade inadequada. A SQA garante que práticas adequadas de gerenciamento de mudanças (Capítulo 22) tenham sido instituídas.

Educação. Toda organização de software quer melhorar suas práticas de engenharia de software. Um fator fundamental para o aperfeiçoamento é a educação dos engenheiros de software, seus gerentes e outros interessados. A organização de SQA assume a liderança no processo de aperfeiçoamento do software (Capítulo 30) e é um proponente fundamental e patrocinador de programas educacionais.

Gerência dos fornecedores. Adquirem-se três categorias de software de fornecedores externos de software — *pacotes prontos, comerciais* (por exemplo, Microsoft Office, oferecidos ao usuário em embalagens), um *shell personalizado* [Hor03] que fornece um esqueleto básico, personalizado de acordo com as necessidades do comprador e *software sob encomenda* que é projetado e construído de forma personalizada a partir de especificações fornecidas pela empresa-cliente. O papel do grupo de SQA é garantir software de alta qualidade por meio da sugestão de práticas específicas de garantia da qualidade que o fornecedor deve (sempre que possível) seguir, e incorporar exigências de qualidade como parte de qualquer contrato com um fornecedor externo.

Administração da segurança. Com o aumento dos crimes cibernéticos e novas regulamentações governamentais referentes à privacidade, toda organização de software deve instituir políticas que protejam os dados em todos os níveis, estabelecer proteção através de firewalls para as aplicações da Internet (WebApps) e garantir que o software não tenha sido alterado internamente, sem autorização. A SQA garante o emprego de processos e tecnologias apropriadas para ter a segurança de software desejada.

Proteção. O fato de o software ser quase sempre um componente fundamental de sistemas que envolvem vidas humanas (por exemplo, aplicações na indústria automotiva ou aeronáutica), o impacto de defeitos ocultos pode ser catastrófico. A SQA pode ser responsável por avaliar o impacto de falhas de software e por iniciar as etapas necessárias para redução de riscos.

“Excelência é a capacidade ilimitada de melhorar a qualidade daquilo que se tem a oferecer.”

Rick Petrin

Administração de riscos. Embora a análise e a redução de riscos (Capítulo 28) seja preocupação dos engenheiros de software, o grupo de SQA garante que as atividades de gestão de riscos sejam conduzidas apropriadamente e que planos de contingência relacionados a riscos tenham sido estabelecidos.

Além de cada uma dessas preocupações e atividades, a SQA trabalha para garantir que atividades de suporte ao software (por exemplo, manutenção, suporte on-line, documentação e manuais) sejam realizadas ou produzidas tendo a qualidade como preocupação dominante.

INFORMAÇÕES



Recursos para Gestão da Qualidade

Há dezenas de recursos para gestão da qualidade disponíveis na Internet, incluindo associações de profissionais, organizações de padrões e fontes de informação genéricas. Veja aqui os sites que são um bom ponto de partida:

American Society for Quality (ASQ) Software Division
www.asq.org/software

Association for Computer Machinery www.acm.org
Data and Analysis Center for Software (DACS)
www.dacs.dtic.mil/

International Organization for Standardization (ISO)
www.iso.ch

ISO SPICE
www.isospice.com

Malcolm Baldrige National Quality Award
www.quality.nist.gov

Software Engineering Institute
www.sei.cmu.edu/

Testes de Software e Engenharia da Qualidade
www.stickyminds.com

Recursos sobre o Seis Sigma
www.isixsigma.com/
www.asq.org/sixsigma/

TickIT International: Tópicos sobre certificação em qualidade
www.tickit.org/international.htm

Gestão da Qualidade Total (TQM, *Total Quality Management*)
Informações gerais:
www.gslis.utexas.edu/~rpollock/tqm.html

Artigos: www.work911.com/tqmarticles.htm

Glossário:
www.quality.org/TQM-MSI/TQM-glossary.html

16.3 TAREFAS, METAS E MÉTRICAS DA SQA

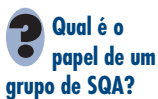
A garantia da qualidade de software é composta por uma série de tarefas associadas a dois elementos distintos — os engenheiros de software que realizam o trabalho técnico e um grupo de SQA que tem a responsabilidade pelo planejamento, supervisão, manutenção de registros, análise e relatórios referentes à garantia da qualidade.

Os engenheiros de software tratam da qualidade (e realizam atividades de controle de qualidade) por meio da aplicação de medidas e métodos técnicos consistentes, conduzindo as revisões técnicas e realizando os bem planejados testes de software.

16.3.1 Tarefas da SQA

A prerrogativa do grupo de SQA é ajudar a equipe de software a obter um produto final de alta qualidade. O SEI (Software Engineering Institute) recomenda um conjunto de ações de SQA que tratam do planejamento, da supervisão, da manutenção de registros, da análise e de relatórios relativos à garantia da qualidade. Essas ações são realizadas (ou facilitadas) por um grupo de SQA independente que:

Prepara um plano de SQA para um projeto. O plano é desenvolvido como parte do planejamento de projeto e é revisado por todos os interessados. As ações de garantia da qualidade realizadas pela equipe de engenharia de software e o grupo de SQA são governados pelo plano, que identifica as avaliações, auditorias e revisões a ser realizadas, padrões que são aplicáveis para o projeto, procedimentos para relatório e acompanhamento de erros,



produtos resultantes produzidos pelo grupo de SQA e o *feedback* que será fornecido à equipe de software.

Participa no desenvolvimento da descrição da gestão de qualidade do projeto. A equipe de software seleciona um processo para o trabalho a ser realizado. O grupo de SQA revisa a descrição de processos para conformidade com a política organizacional, padrões internos de software, padrões impostos externamente (por exemplo, ISO-9001) e outras partes do plano de projeto de software.

Revisa as atividades de engenharia de software para verificar sua conformidade com a gestão de qualidade definida. O grupo de SQA identifica, documenta e acompanha desvios do processo e verifica se as correções foram feitas.

Audita produtos de software resultantes designados para verificar sua conformidade com aqueles definidos como parte da gestão de qualidade. O grupo de SQA revisa produtos resultantes selecionados; identifica, documenta e acompanha os desvios; verifica se as correções foram feitas e, periodicamente, relata os resultados de seu trabalho para o gerente de projeto.

Garante que os desvios no trabalho de software e produtos resultantes sejam documentados e tratados de acordo com um procedimento documentado. Podem ser encontrados desvios no plano de projeto, na descrição do processo, padrões aplicáveis ou no artefato da engenharia de software.

Registra qualquer não aderência e relata ao gerenciamento superior. Itens com problemas (que não atendem às especificações) são acompanhados até que tais problemas sejam resolvidos.

Além dessas ações, o grupo de SQA coordena o controle e o gerenciamento de mudanças (Capítulo 22) e ajuda a coletar e analisar métricas de software.

16.3.2 Metas, atributos e métricas

As ações de SQA descritas na seção anterior são realizadas para atingir um conjunto de metas pragmáticas:

Qualidade dos requisitos. A correção, a completude e a consistência do modelo de requisitos terão forte influência sobre a qualidade de todos os produtos seguintes. A SQA deve assegurar-se de que a equipe de software tenha revisto apropriadamente o modelo de requisitos para a obtenção de um alto nível de qualidade.

Qualidade do projeto. Todo elemento do modelo de projeto deve ser avaliado pela equipe de software para garantir que apresente alta qualidade e que o próprio projeto esteja de acordo com os requisitos. A SQA busca atributos do projeto que sejam indicadores de qualidade.

Qualidade do código. O código-fonte e os produtos relacionados (por exemplo, outras informações descritivas) devem estar em conformidade com os padrões locais de codificação e apresentar características que irão facilitar a manutenção. A SQA deve isolar esses atributos que permitem uma análise razoável da qualidade do código.

Eficácia do controle de qualidade. A equipe de software deve aplicar os recursos limitados de forma a obter a maior probabilidade possível de atingir um resultado de alta qualidade. A SQA analisa a alocação de recursos para revisões e realiza testes para verificar se eles estão ou não sendo alocados da maneira mais efetiva.

A Figura 16.1 (adaptada de [Hya96]) identifica os atributos indicadores da existência de qualidade para cada uma das metas discutidas. As métricas que podem ser utilizadas para indicar a força relativa de um atributo também são mostradas.

"A qualidade jamais é por acidente; ela sempre é o resultado de intenção extraordinária, esforço sincero, direção inteligente e uma hábil execução; representa a escolha inteligente entre muitas alternativas."

William A.
Foster

FIGURA 16.1**Metas, atributos e métricas para qualidade de software***Fonte: Adaptado de (Hya96)*

Meta	Atributo	Métrica
Qualidade das necessidades	Ambiguidade	Número de modificadores ambíguos (por exemplo, muitos, grande, amigável)
	Completeness	Número de TBA, TBD
	Compreensibilidade	Número de seções/subseções
	Volatilidade	Número de mudanças por requisito
		Tempo (por atividade) quando é solicitada a mudança
	Facilidade de atribuição	Número de requisitos não atribuíveis ao projeto/código
	Clareza do modelo	Número de modelos UML
		Número de páginas descritivas por modelo
		Número de erros UML
Qualidade do projeto	Integridade da arquitetura	Existência do modelo da arquitetura
	Completeness dos componentes	Número de componentes que se atribui ao modelo da arquitetura
		Complexidade do projeto procedural
	Complexidade da interface	Número médio de cliques para chegar a uma função ou conteúdo típico
		Apropriabilidade do layout
	Padrões	Número de padrões usados
Qualidade do código	Complexidade	Complexidade ciclométrica
	Facilidade de manutenção	Fatores de projeto (Capítulo 8)
	Compreensibilidade	Porcentagem de comentários internos
		Convenções de atribuição de variáveis
	Reusabilidade	Porcentagem de componentes reutilizados
	Documentação	Índice de legibilidade
Eficiência do controle de qualidade	Alocação de recursos	Porcentagem de horas de pessoal por atividade
	Taxa de completeness	Tempo de finalização real versus previsto
	Eficácia da revisão	Ver métricas de revisão (Capítulo 14)
	Eficácia dos testes	Número de erros encontrados e criticalidade
		Esforço exigido para corrigir um erro
		Origem do erro

16.4 ABORDAGENS FORMAIS DA SQA

Nas seções anteriores, argumentamos que a qualidade de software é tarefa de todos e que pode ser atingida por meio da prática competente de engenharia de software, bem como da aplicação de revisões técnicas, uma estratégia de testes multiescalonados, melhor controle do artefato de software resultante e as mudanças nele feitas e a aplicação dos padrões aceitos de engenharia de software. Além disso, a qualidade pode ser definida em termos de uma ampla gama de atributos de qualidade, e medida (indiretamente) usando uma variedade de índices e métricas.

WebRef

Informações úteis sobre SQA e métodos formais da qualidade podem ser encontradas em www.gslis.utexas.edu/~rpollock/tqm.html.

Ao longo das últimas três décadas, um segmento pequeno, mas eloquente, da comunidade de engenharia de software sustentava que era necessária uma abordagem mais formal para garantir a qualidade de software. Pode-se argumentar que um programa de computador é um objeto matemático. Pode-se definir uma sintaxe e semântica rigorosas para todas as linguagens de programação e está disponível uma rigorosa abordagem da especificação dos requisitos de software (Capítulo 21). Se o modelo de requisitos (especificação) e a linguagem de programação podem ser representados de maneira rigorosa, deve ser possível aplicar uma prova matemática da correção para demonstrar a adequação exata de um programa às suas especificações.

Tentativas de se provar que os programas são corretos não são novas. Dijkstra [Dij76a] e Linger, Mills e Witt [Lin79], entre outros, defenderam provas da correção de programas e associaram-nas ao uso dos conceitos de programação estruturada (Capítulo 10).

16.5 ESTATÍSTICA DA GARANTIA DA QUALIDADE DE SOFTWARE

A estatística da garantia de qualidade reflete uma tendência crescente em toda a indústria de software para tornar mais quantitativa a análise da qualidade. Para software, a estatística da garantia da qualidade implica as seguintes etapas:

Quais são as etapas necessárias para realizar estatística de SQA?

1. Informações sobre erros e defeitos de software são coletadas e classificadas.
2. É feita uma tentativa de associar cada erro e defeito a sua causa subjacente (por exemplo, a não adequação às especificações, erros de projeto, violação de padrões, comunicação inadequada com o cliente).
3. Usando o princípio de Pareto (80% dos defeitos podem ser associados a 20% de todas as possíveis causas), são isoladas os 20% (as *poucas causas vitais*).
4. Assim que as poucas causas vitais tiverem sido identificadas, prossegue-se para a correção dos problemas que provocaram os erros e defeitos.

Esse conceito relativamente simples representa um importante passo para a criação de uma gestão de qualidade adaptativa em que mudanças são feitas para melhorar aqueles elementos do processo que introduzem erros.

16.5.1 Um exemplo genérico

Para ilustrar o uso de métodos estatísticos para a engenharia de software, suponhamos que uma organização de engenharia de software reúna informações sobre erros e defeitos por um período de um ano. Alguns desses erros são revelados à medida que o software é desenvolvido. São encontrados outros (defeitos) após o software ter sido liberado para os usuários finais. Embora centenas de problemas diferentes sejam encontrados, todos podem ser associados a uma (ou mais) das seguintes causas:

- Especificações incompletas ou errôneas (IES, incomplete or erroneous specifications)
- Má interpretação da comunicação do cliente (MCC, misinterpretation of customer communication)
- Desvio intencional das especificações (IDS, intentional deviation from specifications)
- Violação dos padrões de programação (VPS, violation of programming standards)
- Erro na representação de dados (EDR, error in data representation)
- Interface inconsistente de componentes (ICI, inconsistent component interface)
- Erro na lógica de projeto (EDL, error in design logic)
- Testes incompletos ou errôneos (IET, incomplete or erroneous testing)
- Documentação imprecisa ou incompleta (IID, inaccurate or incomplete documentation)
- Erro na tradução do projeto para linguagem de programação (PLT, error in programming language translation of design)
- Interface homem-máquina ambígua ou inconsistente (HCI, ambiguous or inconsistent human/computer interface)
- Outros (MIS, miscellaneous)

“Uma análise estatística, conduzida apropriadamente, é uma dissecação delicada de incertezas, uma cirurgia de suposições.”

M. J. Moroney

"20% do código contém 80% dos erros. Encontre-os, corrija-os!"

Lowell Arthur

Para aplicar a estatística da SQA, é construída a tabela da Figura 16.2. A tabela indica que IES, MCC e EDR são as poucas causas vitais responsáveis por 53% de todos os erros. Deve-se notar, entretanto, que IES, EDR, PLT e EDL seriam escolhidas como as poucas causas vitais se forem considerados apenas os erros graves. Uma vez que forem determinadas as poucas causas vitais, a organização de engenharia de software pode começar a ação corretiva. Por exemplo, para corrigir a MCC, talvez fosse preciso implementar as técnicas de reunião de requisitos (Capítulo 5) para melhorar a qualidade da comunicação do cliente e das especificações. Para melhorar o EDR, pode-se adquirir ferramentas para modelagem de dados e realizar revisões mais rigorosas do projeto de dados.

É importante notar que a ação corretiva se concentra basicamente nas poucas causas vitais. À medida que as poucas causas vitais forem corrigidas, novos candidatos vão para o topo da lista.

As técnicas de estatística da garantia da qualidade para software demonstraram fornecer um aperfeiçoamento substancial da qualidade [Art97]. Em alguns casos, as organizações de software atingiram uma redução de 50% por ano nos defeitos após a aplicação dessas técnicas.

A aplicação de estatística de SQA e o princípio de Pareto podem ser sintetizados em uma única sentença: *Invista seu tempo concentrando-se em coisas que realmente importam, mas, primeiramente, certifique-se de ter entendido aquilo que realmente importa!*

16.5.2 Seis sigma para engenharia de software

Seis Sigma é a estratégia para a estatística da garantia da qualidade mais utilizada na indústria atual. Originalmente popularizada pela Motorola na década de 1980, a estratégia *Seis Sigma* "é uma metodologia rigorosa e disciplinada que usa análise estatística e de dados para medir e melhorar o desempenho operacional de uma empresa através da identificação e da eliminação de defeitos em processos de fabricação e relacionados a serviços" [ISI08]. O termo *Seis Sigma* é derivado de seis desvios-padrão — 3,4 ocorrências (defeitos) por milhão — implicando em um padrão de qualidade extremamente elevado. A metodologia *Seis Sigma* define três etapas essenciais:

- *Definir* as necessidades do cliente e os artefatos passíveis de entrega, bem como as metas de projeto através de métodos bem definidos da comunicação com o cliente.
- *Medir* o processo existente e seu resultado para determinar o desempenho da qualidade atual (reunir métricas para defeitos).
- *Analisar* as métricas para defeitos e determinar as poucas causas vitais.

 **Quais são as etapas essenciais da metodologia Seis Sigma?**

FIGURA 16.2

Coleta de dados para estatística da SQA

Erro	Total		Graves		Moderados		Secundários	
	No.	%	No.	%	No.	%	No.	%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Total	942	100%	128	100%	379	100%	435	100%

Se já existir uma gestão de qualidade, e for necessário um aperfeiçoamento, a estratégia Seis Sigma sugere duas etapas adicionais:

- *Melhorar* o processo por meio da eliminação das causas fundamentais dos defeitos.
- *Controlar* o processo para garantir que trabalhos futuros não reintroduzam as causas dos defeitos.

Essas etapas essenciais e adicionais são, algumas vezes, conhecidas como método DMAIC (**d**efinir, **m**edir, **a**nalisar, **a**perfeiçoar e **c**ontrolar).

Se uma organização estiver desenvolvendo uma gestão de qualidade (e não aperfeiçoando um já existente), nas etapas essenciais são incluídas:

- *Projetar* o processo para: (1) evitar as causas fundamentais dos defeitos e (2) atender as necessidades do cliente.
- *Verificar* se o modelo de processos irá, de fato, evitar defeitos e atender as necessidades do cliente.

Essa variação é algumas vezes denominada método DMADV (**d**efinir, **m**edir, **a**nalisar, **p**rojetar [design] e **v**erificar).

É melhor deixarmos uma discussão completa sobre Seis Sigma para fontes dedicadas ao assunto. Caso tenha maior interesse, veja [ISI08], [Pyz03] e [Sne03].

16.6 CONFIABILIDADE DE SOFTWARE

“O preço inevitável da confiabilidade é a simplicidade.”

C. A. R. Hoare

Não há nenhuma dúvida de que a confiabilidade de um programa de computador é um elemento importante de sua qualidade global. Se um programa falhar frequentemente e repetidas vezes, pouco importa se outros fatores de qualidade de software sejam aceitáveis.

A confiabilidade de software, diferentemente de outros fatores de qualidade, pode ser medida diretamente e estimada usando-se dados históricos e de desenvolvimento. A *confiabilidade de software* é definida em termos estatísticos como “a probabilidade de operação sem falhas de um programa de computador em um dado ambiente por um determinado tempo” [Mus87]. Para ilustrarmos esse conceito, estima-se que o programa X tenha uma confiabilidade de 0,999 depois de decorridas oito horas de processamento. Em outras palavras, se o programa X tiver de que ser executado 1.000 vezes e precisar de um total de oito horas de tempo de processamento (tempo de execução), é provável que ele opere corretamente (sem falhas) 999 vezes.

Toda vez que discutimos confiabilidade de software, surge uma questão fundamental: Qual o significado do termo *falha*? No contexto de qualquer discussão sobre qualidade de software e confiabilidade, falha é a falta de conformidade com os requisitos de software. Mesmo dentro dessa definição existem variações. As falhas podem ser apenas problemáticas ou catastróficas. Uma determinada falha pode ser corrigida em segundos, enquanto outras necessitarão de semanas ou até mesmo meses para serem corrigidas. Para complicar ainda mais o problema, a correção de uma falha pode, na realidade, resultar na introdução de outros erros que resultarão em outras falhas.

16.6.1 Medidas de confiabilidade e disponibilidade

Os primeiros trabalhos sobre confiabilidade de software tentaram extrapolar a matemática da teoria da confiabilidade de hardware para a previsão da confiabilidade de software. A maioria dos modelos de confiabilidade relacionada com hardware tem como base falhas devido a desgaste e não as falhas devido a defeitos de projeto. Em hardware, as falhas devido a desgaste físico (por exemplo, os efeitos decorrentes de temperatura, corrosão, choque) são mais prováveis do que uma falha relacionada ao projeto. Infelizmente, o contrário é verdadeiro para software. Na realidade, todas as falhas de software podem ser associadas a problemas de projeto ou de implementação; o desgaste (ver Capítulo 1) não entra em questão.

Tem havido um debate contínuo sobre a relação entre conceitos-chave na confiabilidade de hardware e sua aplicabilidade ao software. Embora ainda seja preciso estabelecer uma associação

PONTO-CHAVE

Os problemas de confiabilidade de software podem quase sempre ser associados a defeitos de projeto ou de implementação.

PONTO-CHAVE

É importante notar que o MTBF e medidas relacionadas se baseiam em tempo de CPU, e não em tempo de relógio tradicional.



Alguns aspectos da disponibilidade (não discutidos aqui) não têm nada a ver com falha. Por exemplo, programar a parada do sistema (para funções de suporte) faz com que o software fique indisponível.

irrefutável, consideram-se alguns conceitos simples que se aplicam aos elementos de ambos os sistemas.

Se considerarmos um sistema computacional, uma medida de confiabilidade simples é o *tempo médio entre falhas* (MTBF, *mean-time-between-failure*):

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

em que os acrônimos MTTF e MTTR são, respectivamente, *tempo médio para falhar* (*mean-time-to-failure*) e *tempo médio para reparar* (*mean-time-to-repair*)².

Muitos pesquisadores defendem que o MTBF é uma medida mais útil do que quaisquer outras métricas de software relacionadas com a garantia da qualidade discutidas no Capítulo 23. De maneira simples, um usuário final se preocupa com falhas e não com o número total de defeitos. Como cada defeito contido em um programa não tem a mesma taxa de falhas, o número total de defeitos fornece pouca indicação da confiabilidade de um sistema. Por exemplo, considere um programa que esteve em operação por 3.000 horas de processamento sem nenhuma falha. Vários defeitos neste programa podem não ser detectados por dezenas de milhares de horas antes de serem descobertos. O MTBF com esses erros obscuros poderia ser de 30.000 ou até mesmo 60.000 horas de processador. Outros defeitos, embora ainda não descobertos, poderiam ter uma taxa de falhas de 4.000 ou 5.000 horas. Mesmo se cada um dos erros da primeira categoria (aqueles com MTBF longo) fosse eliminado, o impacto sobre a confiabilidade de software seria desprezível.

Entretanto, o MTBF pode ser problemático por duas razões: (1) ele projeta um período de tempo entre falhas, mas não fornece uma projeção da taxa de falhas e (2) o MTBF pode ser mal interpretado como sendo tempo de vida médio, muito embora *não* seja esse o significado.

Uma medida alternativa de confiabilidade é *falha ao longo do tempo* (FIT, *failures-in-time*) — uma medida estatística de quantas falhas um componente terá ao longo de um bilhão de horas de operação. Consequentemente, 1 FIT equivale a uma falha a cada bilhão de horas de operação.

Além de uma medida da confiabilidade, deve-se também desenvolver uma medida de disponibilidade. *Disponibilidade de software* é a probabilidade de que um programa esteja operando de acordo com os requisitos em um dado instante e é definida da seguinte forma:

$$\text{Disponibilidade} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\%$$

A medida de confiabilidade MTBF é igualmente sensível ao MTTF e ao MTTR. A medida de disponibilidade é ligeiramente mais sensível ao MTTR, uma medida indireta da manutenibilidade do software.

16.6.2 Proteção do software

Proteção do software é uma atividade da garantia da qualidade de software que se concentra na identificação e na avaliação de potenciais problemas que podem afetar negativamente um software e provocar falha em todo o sistema. Se os problemas podem ser identificados precocemente na gestão de qualidade, as características para eliminar ou controlar estes problemas podem ser especificadas no projeto de software.

Um processo de modelagem e análise é efetuado como parte de proteção do software. Inicialmente, os problemas são identificados e classificados por criticalidade e risco. Por exemplo, problemas associados a um controle computadorizado de um automóvel podem: (1) provocar uma aceleração descontrolada que não pode ser interrompida, (2) não responder ao acionamento do pedal do breque (através de uma desativação), (3) não operar quando a chave é ativada e (4) perder ou ganhar velocidade lentamente. Uma vez identificados esses perigos no nível de sistema,

“A segurança das pessoas deve ser a maior das leis.”

Cícero

² Embora a depuração (e correções relacionadas) possam ser necessárias como consequência de uma falha, em muitos casos o software funcionará apropriadamente depois de um reinício sem nenhuma outra mudança.

“Não consigo imaginar nenhuma condição que faria com que esse navio viesse a afundar.

A construção naval moderna ultrapassou essa condição.”

E. I. Smith,
capitão do
Titanic

WebRef

Uma proveitosa coleção de artigos sobre proteção de software pode ser encontrada em www.safewareeng.com/.

técnicas de análise são utilizadas para atribuir gravidade e probabilidade de ocorrência.³ Para ser efetivo, o software deve ser analisado no contexto de todo o sistema. Por exemplo, um erro sutil cometido pelo usuário na entrada de dados (as pessoas são componentes do sistema) talvez seja ampliado por uma falha de software e produza dados de controle que posicione de forma inapropriada um dispositivo mecânico. Se e somente se um conjunto de condições ambientais externas for atendido, a posição imprópria do dispositivo mecânico provocará uma falha desastrosa. Análises técnicas [Eri05] como a análise da árvore de falhas, lógica em tempo real e modelos em redes de Petri podem ser usadas para prever a cadeia de eventos que podem causar problemas e a probabilidade que cada um dos eventos irá ocorrer para criar a cadeia.

Uma vez que os problemas são identificados e analisados, os requisitos relacionados com a proteção podem ser especificados para o software. Ou seja, a especificação pode conter uma lista de eventos indesejáveis e as respostas desejadas pelo sistema para esses eventos. O papel do software em administrar eventos indesejáveis é então indicado.

Embora a confiabilidade de software e a proteção do software estejam intimamente relacionadas entre si, é importante entender a diferença sutil entre elas. A confiabilidade de software usa análise estatística para determinar a probabilidade de ocorrência de uma falha de software. Entretanto, a ocorrência de uma falha não resulta, necessariamente, em um problema ou contratempo. A proteção de software examina as maneiras em que falhas resultam em condições que podem levar a um contratempo. Ou seja, as falhas não são consideradas isoladamente, mas sim avaliadas no contexto de todo o sistema computacional e seu ambiente.

Uma discussão completa sobre proteção de software vai além do escopo deste livro. Caso tenha maior interesse no tema proteção de software e questões relacionadas, veja [Smi05], [Dun02] e [Lev95].

16.7 Os Padrões⁴ de Qualidade ISO 9000

Um *sistema de garantia da qualidade* pode ser definido como a estrutura organizacional com responsabilidades, procedimentos, processos e recursos para implementação da gestão da qualidade [ANS87]. Os sistemas de garantia da qualidade são criados para ajudar as organizações a garantir que seus produtos e serviços satisfaçam às expectativas do cliente por meio do atendimento às suas especificações. Tais sistemas cobrem uma grande variedade de atividades englobando todo o ciclo de vida de um produto incluindo planejamento, controle, medições, testes e geração de relatórios e melhorando os níveis de qualidade ao longo de todo o processo de desenvolvimento e fabricação. A ISO 9000 descreve elementos de garantia da qualidade em termos gerais que podem ser aplicados a qualquer empresa, independentemente do tipo de produtos ou serviços.

Para obter a certificação em um dos programas de garantia da qualidade contidos na ISO 9000, as operações e o sistema de qualidade de uma empresa são examinados por auditores independentes, para verificação de sua conformidade ao padrão e operação efetiva. Após aprovação, um organismo representado pelos auditores emite um certificado para a empresa. Auditorias de inspeção semestrais garantem conformidade contínua ao padrão.

As necessidades delineadas pelos tópicos da ISO 9001:2000 são: responsabilidade administrativa, um sistema de qualidade, revisão contratada, controle de projeto, controle de dados e documentos, identificação e rastreabilidade de produtos, controle de processos, inspeções e testes, ações preventivas e corretivas, registros de controle de qualidade, auditorias de qualidade internas, treinamento, manutenção e técnicas estatísticas. Para que uma organização de software seja certificada com a ISO 9001:2000, tem de estabelecer políticas e procedimentos

WebRef

Podemos encontrar uma longa lista de links para recursos relacionados à ISO 9000/9001 em www.tantara.ab.ca/info.htm.

³ Essa abordagem é similar aos métodos de análise de riscos descritos no Capítulo 28. A diferença fundamental é a ênfase em problemas de tecnologia em vez de tópicos relacionados a projeto.

⁴ Essa seção, escrita por Michael Stovsky, foi adaptada de *Fundamentals of ISO 9000*, um livro desenvolvido para *Essential Software Engineering*, um programa de estudos em vídeo desenvolvido pela R. S. Pressman & Associates, Inc. Reimpresso com permissão.

para atender a cada uma das necessidades que acabamos de citar (e outras também) e depois ser capaz de demonstrar que tais políticas e procedimentos estão sendo seguidos. Caso deseje maiores informações sobre a ISO 9001:2000, veja [Ant06], [Mut03] ou [Dob04].



O Padrão ISO 9001:2000

A descrição a seguir define os elementos básicos do padrão ISO 9001:2000.

Informações completas sobre o padrão podem ser obtidas da International Organization for Standardization (www.iso.ch) e outras fontes na Internet (por exemplo, www.praxiom.com).

Estabelecer os elementos de um sistema de gestão da qualidade.

Desenvolver, implementar e aperfeiçoar o sistema.

Definir uma política que enfatize a importância do sistema.

Documentar o sistema de qualidade.

Descrever o processo.

Produzir um manual operacional.

Desenvolver métodos para controlar (atualizar) documentos.

Estabelecer métodos para manutenção de registros.

Dar suporte ao controle e à garantia da qualidade.

Promover a importância da qualidade entre todos os interessados.

Focar na satisfação do cliente.

Definir um plano de qualidade que atenda aos objetivos, às responsabilidades e à autoridade.

Definir mecanismos de comunicação entre os interessados.

Estabelecer mecanismos de revisão para um sistema de gestão da qualidade.

Identificar métodos de revisão e mecanismos de *feedback*.

Definir procedimentos de acompanhamento.

Identificar recursos de qualidade, incluindo elementos de pessoal, treinamento e infraestrutura.

Estabelecer mecanismos de controle.

Para planejamento.

Para as necessidades do cliente.

Para as atividades técnicas (por exemplo, análise, projeto, testes).

Para monitoramento e gerenciamento de projetos.

Definir métodos para reparação.

Avaliar dados e métricas de qualidade.

Definir a abordagem para processos e aperfeiçoamento contínuos da qualidade.

INFORMAÇÕES

16.8 O PLANO DE SQA

O *plano de SQA* fornece um roteiro para instituir a garantia da qualidade de software. Desenvolvido pelo grupo de SQA (ou pela equipe de software, caso não exista um grupo de SQA), o plano serve como um gabarito para atividades de SQA que são instituídas para cada projeto de software.



Gestão da qualidade de software

Objetivo: o objetivo das ferramentas de SQA é ajudar uma equipe de projeto a avaliar e aperfeiçoar a qualidade do artefato de software resultante.

Mecânica: a mecânica das ferramentas varia. Em geral, o intuito é avaliar a qualidade de um artefato específico. Nota: Normalmente são incluídas várias ferramentas para teste de software (ver Capítulos 17 a 20) dentro da categoria de ferramentas para SQA.

Ferramentas representativas:¹

ARM, desenvolvida pela NASA (satc.gsfc.nasa.gov/tools/index.html), fornece medidas que podem ser utilizadas

para avaliar a qualidade de um documento de requisitos de software.

QPR ProcessGuide and Scorecard, desenvolvida pela QPR Software (www.qpronline.com), oferece suporte para Seis Sigma e outras abordagens da gestão de qualidade.

Quality Tools and Templates, desenvolvida pela iSixSigma (www.isixsigma.com/it/), descreve uma ampla gama de ferramentas e métodos úteis para gestão da qualidade.

NASA Quality Resources, desenvolvida pelo Goddard Space Flight Center (sw-assurance.gsfc.nasa.gov/index.php) fornece formulários proveitosos, gabaritos, listas de verificação e ferramentas para SQA.

FERRAMENTAS DO SOFTWARE

¹ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra de ferramentas nessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas por seus respectivos desenvolvedores.

Foi publicado pela IEEE [IEE93] um padrão para planos de SQA. O padrão recomenda uma estrutura que identifique: (1) o propósito e o escopo do plano, (2) uma descrição de todos os artefatos resultantes de engenharia de software (por exemplo, modelos, documentos, código-fonte) que caem dentro do âmbito da SQA, (3) todos os padrões e práticas que são aplicados durante a gestão de qualidade, (4) as ações e tarefas da SQA (incluindo revisões e auditorias) e sua aplicação na gestão de qualidade, (5) as ferramentas e os métodos que dão suporte às ações e tarefas da SQA, (6) procedimentos para administração de configurações de software (Capítulo 22), (7) métodos para montagem, salvaguarda e manutenção de todos os registros relativos à SQA e (8) papéis e responsabilidades dentro da organização relacionados com a qualidade do produto.

16.9 RESUMO

A garantia da qualidade de software é uma atividade universal da engenharia de software que é aplicada a cada etapa da gestão de qualidade. A SQA abrange procedimentos para a aplicação efetiva de métodos e ferramentas, a supervisão de atividades de controle de qualidade como revisões técnicas e testes de software, procedimentos para o gerenciamento de mudanças, procedimentos para garantir a conformidade a padrões, bem como mecanismos para medição e geração de relatórios.

Para realizar a garantia da qualidade de software de forma apropriada, devem ser reunidos, avaliados e disseminados os dados sobre o processo de engenharia de software. A estatística da SQA ajuda a melhorar a qualidade do produto e da própria gestão de qualidade. Os modelos de confiabilidade de software estendem as medidas obtidas, possibilitando que dados coletados de defeitos sejam extrapolados para projeção de taxas de falhas e previsões de confiabilidade.

Em suma, devemos observar as palavras de Dunn e Ullman [Dun82]: “A garantia da qualidade de software é uma tradução dos preceitos administrativos e das disciplinas de projeto da garantia da qualidade para a área gerencial e tecnológica da engenharia de software”. A capacidade de garantir a qualidade é a medida de uma engenharia disciplinada e madura. Quando esse mapeamento é realizado com sucesso, o resultado é uma engenharia de software com maturidade.

PROBLEMAS E PONTOS A PONDERAR

- 16.1.** Algumas pessoas dizem que “o controle das variações é o cerne do controle de qualidade”. Como todo programa que é criado é diferente de qualquer outro programa, quais são as variações que buscamos e como controlá-las?
- 16.2.** É possível avaliar a qualidade de software se o cliente ficar alterando continuamente aquilo que supostamente deveria estar pronto?
- 16.3.** Qualidade e confiabilidade são conceitos relacionados mas são, fundamentalmente, diferentes em uma série de aspectos. Discuta as diferenças.
- 16.4.** Um programa pode ser correto e ainda assim não ser confiável? Explique.
- 16.5.** Um programa pode ser correto e ainda assim não apresentar boa qualidade? Explique.
- 16.6.** Por que normalmente existe tensão entre um grupo de engenharia de software e um grupo de garantia da qualidade de software independente? Isso é salutar?
- 16.7.** Dada a responsabilidade para melhorar a qualidade de software na organização. Qual é a primeira coisa a ser feita? E a seguinte?
- 16.8.** Além da contagem de erros e defeitos, existem outras características contáveis de software que impliquem na qualidade? Quais são, e elas podem ser medidas diretamente?
- 16.9.** O conceito de MTBF para software está sujeito a críticas. Explique o motivo.
- 16.10.** Considere dois sistemas críticos de proteção que são controlados por computador. Enumere pelo menos três perigos para cada um deles que podem ser associados diretamente a falhas de software.

16.11. Adquira uma cópia da ISO 9001:2000 e da ISO 9000-3. Prepare uma apresentação que discuta três necessidades da ISO 9001 e como elas se aplicam no contexto de software.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Livros como os de Hoyle (*Quality Management Fundamentals*, Butterworth-Heinemann, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), El Emam (*The ROI from Software Quality*, Auerbach, 2005), Horch (*Practical Guide to Software Quality Management*, Artech House, 2003) e Nance e Arthur (*Managing Software Quality*, Springer, 2002) são excelentes apresentações em termos gerenciais dos benefícios dos programas formais para garantia da qualidade de software. Livros como os de Deming [Dem86], Juran (*Juran on Quality by Design*, Free Press, 1992) e Crosby [Cro79] e *Quality Is Still Free*, McGraw-Hill, 1995) não se concentram em software, mas são leituras obrigatórias para gerentes seniores com responsabilidade pelo desenvolvimento de software. Gluckman e Roome (*Everyday Heroes of Quality Movement*, Dorset House, 1993) humaniza as questões da qualidade contando a história dos participantes em um processo de qualidade. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) apresenta uma visão quantitativa da qualidade de software.

Livros como os de Evans (*Total Quality: Management, Organization and Strategy*, 4. ed., South-Western College Publishing, 2004), Bru (*Six Sigma for Managers*, McGraw-Hill, 2005) e Dobb (*ISO 9001:2000 Quality Registration Step-by-Step*, 3. ed., Butterworth-Heinemann, 2004) são representativos dos muitos livros escritos sobre, respectivamente, TQM, Seis Sigma e ISO 9001:2000.

Pham (*System Software Reliability*, Springer, 2006), Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, 2. ed., McGraw-Hill, 2004) e Peled (*Software Reliability Methods*, Springer, 2001) escreveram guias práticos que descrevem métodos para medir e analisar a confiabilidade de software.

Vincoli (*Basic Guide to System Safety*, Wiley, 2006), Dhillon (*Engineering Safety*, World Scientific Publishing Co., Inc., 2003), Hermann (*Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000), Storey (*Safety-Critical Computer Systems*, Addison-Wesley, 1996) e Leveson [Lev95] são as discussões mais abrangentes sobre segurança de software e sistemas publicadas até hoje. Além desses, van der Meulen (*Definitions for Hardware and Software Safety Engineers*, Springer-Verlag, 2000) oferece um compêndio completo de importantes conceitos e termos de confiabilidade e proteção; Gartner (*Testing Safety-Related Software*, Springer-Verlag, 1999) é um guia especializado para testar sistemas de proteção críticos; Friedman e Voas (*Software Assessment: Reliability Safety and Testability*, Wiley, 1995) fornecem modelos úteis para avaliar a confiabilidade e a proteção. Ericson (*Hazard Analysis Techniques for System Safety*, Wiley, 2005) tratam da área cada vez mais importante da análise de riscos.

Uma ampla gama de fontes de informação sobre garantia da qualidade de software encontra-se à disposição na Internet. Uma lista atualizada de referências que são relevantes para a SQA pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

depuração.....	421
grupo de teste independente	403
revisão de configuração.....	417
teste alfa.....	417
teste beta.....	417
teste de classe ...	415
teste de disponibilização... ..	420
teste de integração	409
teste de regressão.....	416
teste de sistema ..	418
teste de validação	416
teste unitário	407
V&V.....	402

A estratégia de teste de software fornece um roteiro que descreve os passos a ser executados como parte do teste, define quando esses passos são planejados e então executados, e quanto trabalho, tempo e recursos serão necessários. Portanto, qualquer estratégia de teste deve incorporar planejamento dos testes, projeto de casos de teste, execução dos testes, coleta e avaliação dos dados resultantes.

Uma estratégia de teste de software deve ser flexível o bastante para promover uma abordagem de teste personalizada. Ao mesmo tempo, deve ser rígida o bastante para estimular um planejamento razoável e o acompanhamento, à medida que o projeto progride. Shooman [Sho83] discute essas questões:

De muitas formas, o teste é um processo individualista e o número de tipos diferentes de testes varia tanto quanto as diferentes abordagens de desenvolvimento. Por muitos anos, nossa única defesa contra os erros de programação era um projeto cuidadoso e a inteligência do programador. Estamos agora em uma era na qual as modernas técnicas de projeto [e revisões técnicas] estão nos ajudando a reduzir a quantidade de erros iniciais inerentes ao código. De maneira semelhante, diferentes métodos de teste estão começando a se agrupar em várias abordagens e filosofias distintas.

Essas “abordagens e filosofias” são chamadas de *estratégia* — o tópico que será apresentado neste capítulo. Nos Capítulos 18 a 20, apresentamos os métodos e técnicas de teste que implementam a estratégia.

PANORAMA

O que é? O software é testado para revelar erros cometidos inadvertidamente quando projetado e construído. Mas como conduzir os testes? Devemos estabelecer um plano formal para nossos testes? Devemos testar o programa como um todo ou executar testes somente em uma parte dele? Devemos refazer os testes quando acrescentamos novos componentes ao sistema? Quando devemos envolver o cliente? Essas e muitas outras questões são respondidas quando desenvolvemos uma estratégia de teste de software.

Quem realiza? Uma estratégia para teste de software é desenvolvida pelo gerente de projeto, pelos engenheiros de software e pelos especialistas em testes.

Porque é importante? O teste muitas vezes requer mais trabalho de projeto do que qualquer outra ação da engenharia de software. Se for feito casualmente, perde-se tempo, fazem-se esforços desnecessários, e, ainda pior, erros passam sem ser detectados. Portanto, é razoável estabelecer uma estratégia sistemática para teste de software.

Quais são as etapas envolvidas? O teste começa pelo “pequeno” e passa para o “grande”. Ou

seja, os testes iniciais focalizam um único componente ou um pequeno grupo de componentes relacionados e aplicam-se testes para descobrir erros nos dados e na lógica de processamento que foram encapsulados pelo(s) componente(s). Depois de testados, os componentes devem ser integrados até que o sistema completo esteja pronto. Nesse ponto, são executados muitos testes de ordem superior para descobrir erros ao atender aos requisitos do cliente. À medida que os erros forem descobertos, devem ser diagnosticados e corrigidos usando um processo chamado de depuração.

Qual é o artefato? A especificação do teste documenta a abordagem da equipe de software para o teste, definindo um plano que descreve uma estratégia global e um procedimento designando etapas específicas de teste e os tipos de testes que serão feitos.

Como garantir que o trabalho foi feito corretamente? Revisando a especificação do teste antes do teste, pode-se avaliar a integralidade dos casos de testes e das tarefas de teste. Um plano e um procedimento de teste eficaz levarão a uma construção ordenada do software e à descoberta de erros em cada estágio do processo de construção.

17.1 UMA ABORDAGEM ESTRATÉGICA DO TESTE DE SOFTWARE

Teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente. Por essa razão, deverá ser definido para o processo de software um modelo (*template*) para o teste — um conjunto de etapas no qual pode-se colocar técnicas específicas de projeto de caso de teste e métodos de teste.

Muitas estratégias de teste de software já foram propostas na literatura. Todas elas fornecem um modelo para o teste e todas têm as seguintes características genéricas:

- Para executar um teste eficaz, proceder a revisões técnicas eficazes (Capítulo 15). Fazendo isso, muitos erros serão eliminados antes do começo do teste.
- O teste começa no nível de componente e progride em direção à integração do sistema computacional como um todo.
- Diferentes técnicas de teste são apropriadas para diferentes abordagens de engenharia de software e em diferentes pontos no tempo.
- O teste é feito pelo desenvolvedor do software e (para grandes projetos) por um grupo independente de teste.
- O teste e a depuração são atividades diferentes, mas a depuração deve ser associada com alguma estratégia de teste.

WebRef

Recursos úteis para o teste de software podem ser encontrados em www.mtsu.edu/~storm/.

Uma estratégia de teste de software deve acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem como testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente. Uma estratégia deverá fornecer diretrizes para o profissional e uma série de metas para o gerente. Devido ao fato de os passos da estratégia de teste ocorrerem no instante em que as pressões pelo prazo começam a aumentar, deve ser possível medir o progresso no desenvolvimento e os problemas devem ser revelados o mais cedo possível.

17.1.1 Verificação e validação

O teste de software é um elemento de um tópico mais amplo, muitas vezes conhecido como verificação e validação (V&V). *Verificação* refere-se ao conjunto de tarefas que garantem que o software implementa corretamente uma função específica. *Validação* refere-se a um conjunto de tarefas que asseguram que o software foi criado e pode ser rastreado segundo os requisitos do cliente. Boehm [Boe81] define de outra maneira:

Verificação: “Estamos criando o produto corretamente?”

Validação: “Estamos criando o produto certo?”

A definição de V&V abrange muitas atividades de garantia da qualidade do software (Capítulo 16).¹

A verificação e a validação incluem uma ampla gama de atividades de SQA (*software quality assurance*, garantia da qualidade de software): revisões técnicas, auditorias de qualidade e configuração, monitoramento de desempenho, simulação, estudo de viabilidade, revisão de documentação, revisão de base de dados, análise de algoritmo, teste de desenvolvimento, teste de usabilidade, teste de qualificação, teste de aceitação e teste de instalação. Embora a aplicação de teste tenha um papel extremamente importante em V&V, muitas outras atividades também são necessárias.

O teste proporciona o último elemento a partir do qual a qualidade pode ser estimada e, mais pragmaticamente, os erros podem ser descobertos. Mas o teste não deve ser visto como uma rede de segurança. Como se costuma dizer, “Você não pode testar qualidade. Se a qualidade não

“O teste é uma parte inevitável de qualquer trabalho responsável para o desenvolvimento de um sistema de software.”

William Howden



Não seja tolo e não considere o teste como uma “rede de segurança” que detectará todos os erros ocorridos devido a práticas deficientes de engenharia de software. Isso não acontecerá. Enfatize a qualidade e a detecção de erro em todo o processo de software.

¹ Deve-se observar que há uma forte divergência de opinião sobre quais tipos de testes constituem “validação”. Algumas pessoas acreditam que *todo* o teste é verificação e que a validação é executada quando os requisitos são examinados e aprovados, e mais tarde pelo usuário, com o sistema já em operação. Outras pessoas consideram o teste de unidade e de integração (Seções 17.3.1 e 17.3.2) como verificação e o teste de ordem superior (Seções 17.6 e 17.7) como validação.

está lá antes de um teste, ela não estará lá quando o teste terminar”. A qualidade é incorporada ao software através do processo de engenharia de software. A aplicação correta de métodos e ferramentas, de revisões técnicas eficazes, e de um sólido gerenciamento e avaliação conduzem todos à qualidade que é confirmada durante o teste.

Miller [Mil77] relaciona teste de software com garantia de qualidade dizendo que “a motivação que está por trás do teste de programas é a confirmação da qualidade de software com métodos que podem ser economicamente e efetivamente aplicados a todos os sistemas, de grande e pequena escala”.

17.1.2 Organizando o teste de software

Para todo projeto de software, há um conflito de interesses inerente que ocorre logo que o teste começa. As pessoas que criaram o software são agora convocadas para testá-lo. Isso parece essencialmente inofensivo; afinal, quem conhece o programa melhor do que os seus próprios desenvolvedores? Infelizmente, esses mesmos desenvolvedores têm interesse em demonstrar que o programa é isento de erros e funciona de acordo com os requisitos do cliente, e também será concluído dentro do prazo e do orçamento previsto. Cada um desses interesses vai contra o teste completo.

De um ponto de vista psicológico, a análise e o projeto de software (juntamente com a sua codificação) são tarefas construtivas. O engenheiro de software analisa, modela e então cria um programa de computador e sua documentação. Como qualquer outro construtor, o engenheiro de software tem orgulho do edifício que ele construiu e encara com desconfiança qualquer um que tenta estragar sua obra. Quando o teste começa, há uma tentativa sutil, embora definida de “quebrar” aquela coisa que o engenheiro de software construiu. Do ponto de vista do construtor, o teste pode ser considerado como (psicologicamente) destrutivo. Assim, o construtor vai, calmamente, projetando e executando testes que demonstram que o programa funciona, em vez de descobrir os erros. Infelizmente, os erros estão presentes. E se o engenheiro de software não os encontrar, o cliente encontrará!

Frequentemente, há muitas noções incorretas que podem ser inferidas erroneamente a partir da discussão apresentada: (1) que o desenvolvedor de software não deve fazer nenhum teste, (2) que o software deve ser “atirado aos leões”, ou seja, entregue a estranhos que realizarão testes implacáveis, (3) que os testadores se envolvem com o projeto somente no início das etapas do teste. Todas essas afirmativas são incorretas.

O desenvolvedor do software é sempre responsável pelo teste das unidades individuais (componentes) do programa, garantindo que cada uma execute a função ou apresente o comportamento para o qual foi projetada. Em muitos casos, o desenvolvedor faz também o teste de integração — uma etapa de teste que leva à construção (e teste) da arquitetura completa do software. Somente depois que a arquitetura do software foi completada é que o grupo de teste independente se envolve.

O papel de um *grupo independente de teste* (ITG, *independent test group*) é remover problemas inerentes associados com o fato de deixar o criador testar a coisa que ele mesmo criou. O teste independente remove o conflito de interesses que, de outra forma, poderia estar presente. Afinal, o pessoal de ITG é pago para encontrar erros.

No entanto, você não entrega simplesmente o programa para o pessoal do ITG e vai embora. O desenvolvedor e o pessoal do ITG trabalham juntos durante todo o projeto de software para garantir que testes completos sejam realizados. Enquanto o teste está sendo realizado, o desenvolvedor deve estar disponível para corrigir os erros encontrados.

O ITG faz parte da equipe de desenvolvimento de software no sentido de que ele se envolve durante a análise, e o projeto e permanece envolvido (planejando e especificando procedimentos de teste) durante o projeto inteiro. No entanto, em muitos casos o ITG se reporta à organização de garantia de qualidade do software, adquirindo assim um grau de independência que poderia não ser possível se fizesse parte da organização de engenharia de software.

“Otimismo é a doença ocupacional da programação; o teste é seu tratamento.”

Kent Beck

PONTO-CHAVE

Um grupo independente de teste não tem o “conflito de interesses” que os criadores de software podem ter.

“O primeiro erro que as pessoas cometem é pensar que a equipe de teste é responsável por garantir a qualidade.”

Brian Marick

17.1.3 Estratégia de teste de software — a visão ampla

O processo de software pode ser visto como a espiral ilustrada na Figura 17.1. Inicialmente, a engenharia de sistemas define o papel do software e passa à análise dos requisitos de software, onde são estabelecidos o domínio da informação, função, comportamento, desempenho, restrições e critérios de validação para o software. Deslocando-se para o interior da espiral, chega-se ao projeto e finalmente à codificação. Para desenvolver software de computador, percorre-se a espiral para o interior (sentido anti-horário) ao longo de linhas que indicam a diminuição do nível de abstração em cada volta.

Uma estratégia para teste de software pode também ser vista no conceito da espiral (Figura 17.1). O *teste de unidade* começa no centro da espiral e se concentra em cada unidade (por exemplo: componente, classe, ou objeto de conteúdo de WebApp) do software conforme implementado no código-fonte. O teste prossegue movendo-se em direção ao exterior da espiral, passando pelo *teste de integração*, em que o foco está no projeto e construção da arquitetura de software. Continuando na mesma direção da espiral, encontramos o *teste de validação*, em que requisitos estabelecidos como parte dos requisitos de modelagem são validados em relação ao software criado. Finalmente, chegamos ao *teste do sistema*, no qual o software e outros elementos são testados como um todo. Para testar um software de computador, percorre-se a espiral em direção ao seu exterior, no sentido horário, ao longo de linhas que indicam o escopo do teste a cada volta.

Considerando o processo de um ponto de vista procedimental, o teste dentro do contexto de engenharia de software é na realidade uma série de quatro etapas que são implementadas sequencialmente. As etapas estão ilustradas na Figura 17.2. Inicialmente, os testes focalizam cada componente individualmente, garantindo que ele funcione adequadamente como uma unidade, daí o nome *teste de unidade*. O teste de unidade usa intensamente técnicas de teste com caminhos específicos na estrutura de controle de um componente para garantir a cobertura completa e a máxima detecção de erro. Em seguida, o componente deve ser montado ou integrado para formar o pacote completo de software. O *teste de integração* cuida de problemas associados com aspectos duais de verificação e construção de programa. Técnicas de projeto de casos de teste que focalizam em entradas e saídas são mais predominantes durante a integração, embora técnicas que usam caminhos específicos de programa possam ser utilizadas para segurança dos principais caminhos de controle. Depois que o software foi integrado (construído), é executada uma série de *testes de ordem superior*. Os critérios de validação (estabelecidos durante a análise de requisitos) devem ser avaliados. O *teste de validação* proporciona a garantia final de que o software satisfaz a todos os requisitos informativos, funcionais, comportamentais e de desempenho.

A última etapa de teste de ordem superior extrapola os limites da engenharia de software, entrando em um contexto mais amplo de engenharia de sistemas de computadores. O software, uma vez validado, deve ser combinado com outros elementos do sistema (por exemplo, hardware, pessoas, base de dados). O *teste de sistema* verifica se todos os elementos se combinam corretamente e se a função/desempenho global do sistema é conseguida.

? Qual é a estratégia global para teste de software?

WebRef

Recursos úteis para testadores de software podem ser encontrados em www.SQAtester.com.

FIGURA 17.1
Estratégia de teste

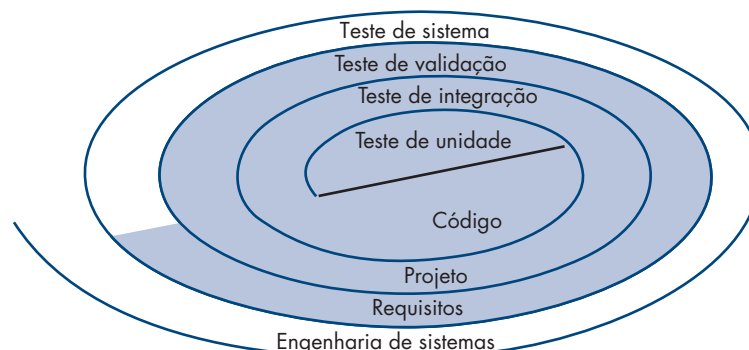
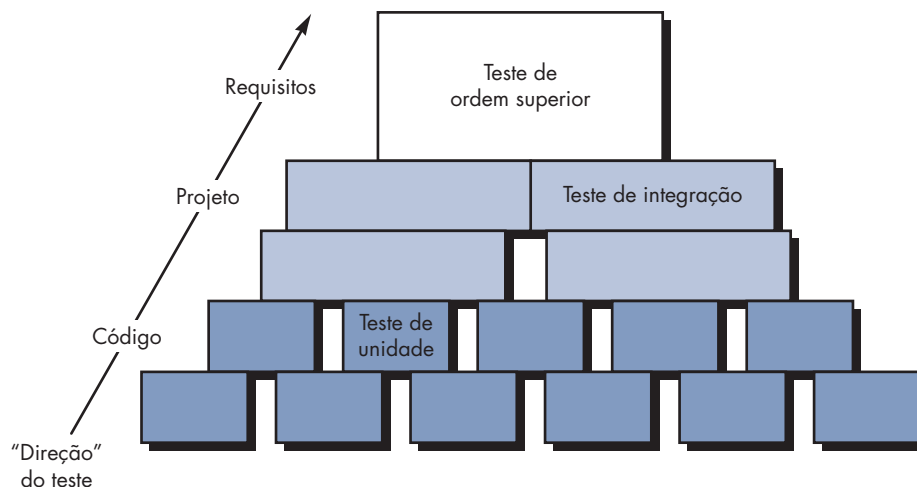


FIGURA 17.2

Etapas de teste de software



CASA SEGURA



Preparando-se para o teste

Cena: No escritório de Doug Miller, quando o projeto no nível de componente está em andamento e começa-se a construção de certos componentes.

Participantes: Doug Miller, gerente de engenharia de software, Vinod, Jamie, Ed e Shakira — membros da equipe de engenharia de software CasaSegura.

Conversa:

Doug: Parece-me que não dedicamos tempo suficiente para falar sobre o teste.

Vinod: É verdade, mas nós estávamos todos um tanto ocupados. Além disso, estivemos pensando sobre isso... Na verdade, mais do que pensando.

Doug (rindo): Eu sei... Todos nós estamos sobrecarregados, mas ainda temos de pensar adiante.

Shakira: Eu gosto da ideia de projetar testes de unidades antes de começar a codificar qualquer um de meus componentes, e é o que estou tentando fazer. Tenho um arquivo grande de testes a ser executados logo que o código dos meus componentes estiver completo.

Doug: Esse é o conceito de Extreme Programming [um processo ágil de desenvolvimento de software; veja o Capítulo 3] não é mesmo?

Ed: É isso mesmo. Apesar de não estarmos usando Extreme Programming diretamente, decidimos que seria uma boa ideia projetar testes unitários antes de criar o componente — o projeto nos dá todas as informações de que precisamos.

Jamie: Eu já fiz a mesma coisa.

Vinod: E eu já assumi o papel de integrador, de forma que, todas as vezes que um dos rapazes passar um componente para mim, eu vou integrá-lo e executar uma série de testes de regressão no programa parcialmente integrado. Estive trabalhando para projetar uma série de testes apropriados para cada função no sistema.

Doug (para Vinod): Com que frequência você fará os testes?

Vinod: Todos os dias... Até que o sistema esteja integrado... Bem, quero dizer, até que o incremento de software que pretendemos fornecer esteja integrado.

Doug: Rapazes, vocês estão mais adiantados do que eu!

Vinod (rindo): Antecipação é tudo no negócio de software, chefe.

17.1.4 Critérios para conclusão do teste

Uma questão clássica surge todas as vezes que se discute teste de software: “Quando podemos dizer que terminamos os testes — como podemos saber que já testamos o suficiente?”. Infelizmente, não há uma resposta definitiva para essa pergunta, mas há algumas respostas pragmáticas e algumas tentativas iniciais e empíricas.

Quando finalizamos o teste?

Uma resposta é: “O teste nunca termina; o encargo simplesmente passa do engenheiro de software para o usuário final”. Todas as vezes que o usuário executa o programa no computador, o programa está sendo testado. Esse fato destaca a importância de outras atividades de garantia de qualidade do software. Outra resposta (um tanto cínica, mas ainda assim, exata) é: “O teste acaba quando o tempo ou o dinheiro acabam”.

Embora alguns profissionais possam argumentar a respeito dessas respostas, o fato é que é necessário um critério mais rigoroso para determinar quando já foram executados testes em número suficiente. A abordagem *engenharia de software de sala limpa* (Capítulo 21) sugere técnicas de uso estatístico [Kel00] que executa uma série de testes derivados de uma amostragem estatística de todas as execuções possíveis do programa por todos os usuários em uma população escolhida. Outros (por exemplo, [Sin99]) defendem o uso de modelagem estatística e teoria de confiabilidade de software para prever a integralidade do teste.

Coletando métricas durante o teste do software e utilizando modelos existentes de confiabilidade de software, é possível desenvolver diretrizes significativas para responder à questão: “Quando terminamos o teste?”. Há uma pequena controvérsia de que ainda resta trabalho a ser feito até que sejam estabelecidas regras quantitativas para o teste, mas as abordagens empíricas que existem no momento são consideravelmente melhores do que a simples intuição.

WebRef

Um glossário claro de termos de teste pode ser encontrado em www.testingstandards.co.uk/living_glossary.htm.

17.2 PROBLEMAS ESTRATÉGICOS

Mais adiante neste capítulo, apresentamos uma estratégia sistemática para teste de software. Mas até mesmo a melhor estratégia fracassará se não for resolvida uma série de problemas e obstáculos. Tom Gilb [Gil95] argumenta que uma estratégia de teste de software terá sucesso quando os testadores de software:

Quais as diretrizes que levam a uma estratégia de teste de software bem-sucedida?

Especifiquem os requisitos do produto de uma maneira quantificável, muito antes de começar o teste. Embora o objetivo principal do teste seja encontrar erros, uma boa estratégia de teste também avalia outras características de qualidade como portabilidade, possibilidade de manutenção e utilidade (Capítulo 14). Esses requisitos devem ser especificados de forma que sejam mensuráveis de tal forma que os resultados do teste não sejam ambíguos.

Definirem explicitamente os objetivos do teste. Os objetivos específicos do teste deverão ser definidos em termos mensuráveis. Por exemplo, a eficiência do teste, a abrangência do teste, o tempo médio entre falhas, o custo para localizar e corrigir defeitos, densidade de defeitos restantes ou frequência de ocorrência e horas de trabalho em testes deverão ser definidos dentro do plano de teste.

Entenderem os usuários do software e desenvolverem um perfil para cada categoria de usuário. Casos de uso que descrevem o cenário de interação para cada classe de usuário podem reduzir o trabalho global de teste, focalizando o teste sobre a utilização real do produto.

Desenvolverem um plano de teste que enfatize o “teste do ciclo rápido”. Gilb [Gil95] recomenda que uma equipe de software “aprenda a testar em ciclos rápidos (2% do trabalho de projeto) de incrementos de funcionalidade e/ou melhora da qualidade úteis ao cliente, ou pelo menos passíveis de experimentação no campo”. O retorno gerado por esses testes de ciclo rápido pode ser usado para controlar os níveis de qualidade e as correspondentes estratégias de teste.

Criarem software “robusto” que seja projetado para testar-se a si próprio. O software deverá ser projetado de maneira a usar técnicas “antidefeitos” (Seção 17.3.1). Isto é, o software deve ser capaz de diagnosticar certas classes de erros. Além disso, o projeto deverá acomodar teste automático e teste de regressão.

Usarem revisões técnicas eficazes como filtro antes do teste. As revisões técnicas (Capítulo 15) podem ser tão eficazes quanto o teste para descobrir erros. Por essa razão, as revisões podem reduzir o trabalho de teste necessário para produzir software de alta qualidade.

WebRef

Uma excelente lista de recursos de teste pode ser encontrada em www.io.com/~wazmo/qa/.

“Teste que leva em conta somente os requisitos do usuário final é como a inspeção de um edifício baseado apenas no trabalho executado pelo projetista de interiores, não considerando as fundações, vigas e tubulações.”

Boris Beizer

Executarem revisões técnicas para avaliar a estratégia de teste e os próprios casos de testes. As revisões técnicas podem descobrir inconsistências, omissões e erros graves na abordagem de teste. Isso poupa tempo e também melhora a qualidade do produto.

Desenvolverem abordagem de melhora contínua para o processo de teste. A estratégia de teste deverá ser medida. As métricas coletadas durante o teste deverão ser usadas como parte de uma abordagem de controle estatístico de processo para o teste de software.

17.3 ESTRATÉGIAS PARA SOFTWARE CONVENCIONAL²

Há muitas estratégias que podem ser utilizadas para testar um software. Em um dos extremos, pode-se esperar até que o sistema esteja totalmente construído e então executar os testes no sistema completo esperando encontrar os erros. Essa abordagem, embora atraente, simplesmente não funciona; resultará em um software defeituoso que desagrade todos os que investiram nele. No outro extremo, você pode executar testes diariamente, sempre que uma parte do sistema seja construída. Essa abordagem, embora menos atraente, pode ser muito eficaz. Infelizmente, alguns desenvolvedores de software hesitam em usá-la. O que fazer?

Uma estratégia de teste que é preferida pela maioria das equipes de software está entre os dois extremos. Ela assume uma visão incremental do teste, começando com o teste das unidades individuais de programa, passando para os testes destinados a facilitar a integração de unidades e culminando com testes que usam o sistema concluído. Cada uma dessas classes de testes é descrita nas próximas seções.

17.3.1 Teste de unidade

O teste de unidade focaliza o esforço de verificação na menor unidade de projeto do software — o componente ou módulo de software. Usando como guia a descrição de projeto no nível de componente, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo. A complexidade relativa dos testes e os erros que revelam são limitados pelo escopo restrito estabelecido para o teste de unidade. Esse teste enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Esse tipo de teste pode ser conduzido em paralelo para diversos componentes.



Não é má ideia projetar casos de testes de unidade antes de desenvolver o código para um componente. É bom assegurar-se que você desenvolverá código que passará nos testes.



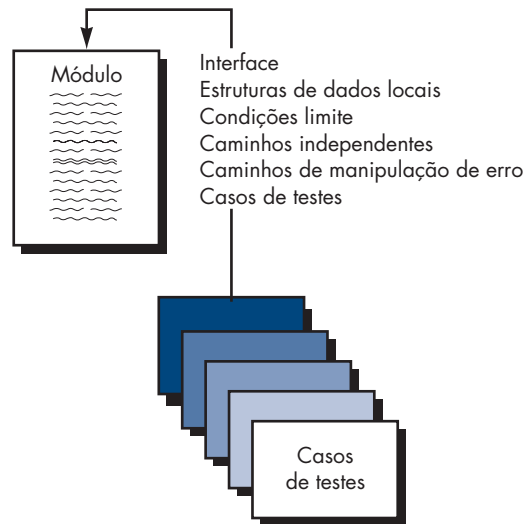
Quais os erros mais comuns encontrados durante o teste de unidade?

Considerações do teste de unidade. Os testes de unidade estão ilustrados esquematicamente na Figura 17.3. A interface do módulo é testada para assegurar que as informações fluam corretamente para dentro e para fora da unidade de programa que está sendo testada. A estrutura de dado local é examinada para garantir que os dados armazenados temporariamente mantenham sua integridade durante todos os passos na execução de um algoritmo. Todos os caminhos independentes da estrutura de controle são usados para assegurar que todas as instruções em um módulo tenham sido executadas pelo menos uma vez. As condições limite são testadas para garantir que o módulo opere adequadamente nas fronteiras estabelecidas para limitar ou restringir o processamento. Finalmente, são testados todos os caminhos de manipulação de erro.

O fluxo de dados por meio da interface de um componente é testado antes de iniciar qualquer outro teste. Se os dados não entram e saem corretamente, todos os outros testes são discutíveis. Além disso, estruturas de dados locais deverão ser ensaiadas e o impacto local sobre dados globais deve ser apurado (se possível) durante o teste de unidade.

O teste seletivo de caminhos de execução é uma tarefa essencial durante o teste de unidade. Casos de testes deverão ser projetados para descobrir erros devido a computações errôneas, comparações incorretas ou fluxo de controle inadequado.

² Neste livro, utilizamos os termos *software convencional* ou *software tradicional* para nos referir às arquiteturas de software do tipo hierárquico comum ou chamada-e-retorno que são frequentemente encontradas em uma variedade de domínios de aplicações. Arquiteturas de software tradicionais *não* são orientadas a objeto e não abrangem WebApps.

FIGURA 17.3**Teste de unidade**

O teste de fronteira é uma das tarefas mais importantes do teste de unidade. O software frequentemente falha nas suas fronteiras. Isto é, os erros frequentemente ocorrem quando o n -ésimo elemento de um conjunto n -dimensional é processado, quando a i -ésima repetição de um laço com i passadas é chamada, quando o valor máximo ou mínimo permitido é encontrado. Casos de testes que utilizam estrutura de dados, fluxo de controle e valores de dados logo abaixo, iguais, ou logo acima dos máximos e mínimos têm grande possibilidade de descobrir erros.

Um bom projeto prevê condições de erro e estabelece caminhos de manipulação de erro para redirecionar ou encerrar ordenadamente o processamento quando ocorre um erro. Yourdon [You75] chama essa abordagem de *antidefeitos*. Infelizmente, há uma tendência de incorporar manipulação de erro no software e nunca testá-lo. Uma história real pode servir como ilustração:

Um sistema de projeto auxiliado por computador (CAD) foi desenvolvido sob contrato. Em um módulo de processamento de transação, um programador brincalhão colocou a seguinte mensagem de manipulação de erro após uma série de testes condicionais que chamavam vários desvios de fluxo de controle: ERRO! VOCÊ NÃO TEM SAÍDA AQUI. Essa “mensagem de erro” foi descoberta por um cliente durante o treinamento com um usuário!

Entre os erros potenciais que deverão ser testados quando a manipulação de erro é avaliada estão: (1) descrição confusa do erro, (2) o erro apontado não corresponde ao erro encontrado, (3) a condição do erro causa intervenção do sistema antes da manipulação do erro, (4) o processamento exceção-condição é incorreto, ou (5) a descrição do erro não fornece informações suficientes para ajudar na localização da causa do erro.

Procedimentos de teste de unidade. O teste de unidade normalmente é considerado um auxiliar para a etapa de codificação. O projeto dos testes de unidade pode ocorrer antes de começar a codificação ou depois que o código-fonte tiver sido gerado. Um exame das informações de projeto fornece instruções para estabelecer casos de testes que provavelmente mostrarão os erros em cada uma das categorias descritas anteriormente. Cada caso de teste deverá ser acooplado com um conjunto de resultados esperados.

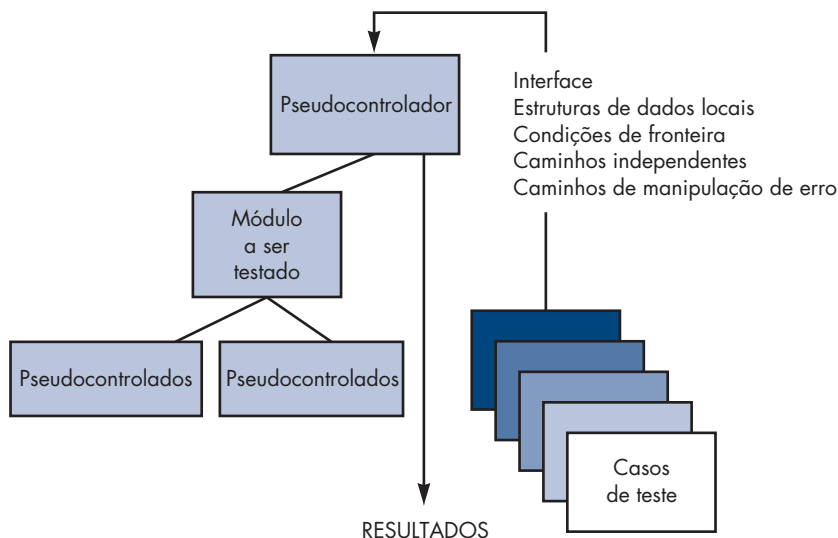
Devido ao fato de um componente não ser um programa independente (*stand-alone*), deve ser desenvolvido um pseudocontrolador (*driver*) e/ou um pseudocontrolado (*stub*) para cada teste de unidade. O ambiente de teste de unidade está ilustrado na Figura 17.4. Em muitas aplicações, um *pseudocontrolador* nada mais é do que um “programa principal” que aceita dados

WebRef

Informações úteis sobre uma ampla variedade de artigos e recursos para “teste ágil” podem ser encontradas em testing.com/agile/.



Projete testes para executar todos os caminhos de manipulação de erro. Se não fizer isso, o caminho pode falhar quando for solicitado, piorando uma situação já ruim.

FIGURA 17.4**Ambiente de teste de unidade**

Há algumas situações nas quais você não terá recursos para fazer um teste de unidade simples. Selecione os módulos críticos ou complexos e faça o teste de unidade apenas neles.

do caso de teste, passa esses dados para o componente (a ser testado), e imprime resultados relevantes. Os *pseudocontroladores* servem para substituir módulos que são subordinados (chamados pelo) componente a ser testado. Um pseudocontrolado, ou “pseudosubprograma”, usa a interface dos módulos subordinados, pode fazer uma manipulação de dados mínima, fornece uma verificação de entrada e retorna o controle para o módulo que está sendo testado.

Pseudocontroladores e pseudocontrolados representam despesas indiretas. Isto é, ambos são softwares que devem ser escritos (projeto formal normalmente não é aplicado), mas que não são fornecidos com o produto de software final. Se os pseudocontroladores e pseudocontrolados são mantidos simples, as despesas reais indiretas são relativamente baixas. Infelizmente, muitos componentes não podem ser adequadamente testados no nível de unidade de modo adequado com software adicional simples. Em tais casos, o teste completo pode ser adiado até a etapa de integração (em que os pseudocontroladores e pseudocontrolados são também usados).

O teste de unidade é simplificado quando um componente com alta coesão é projetado. Quando somente uma função é implementada por um componente, o número de casos de teste é reduzido, e os erros podem ser mais facilmente previstos e descobertos.

17.3.2 Teste de integração

Um novato no mundo do software pode levantar uma questão aparentemente legítima quando todos os módulos tiverem passado pelo teste de unidade: “Se todos funcionam individualmente, porque você duvida que funcionem quando estiverem juntos?”. O problema, naturalmente, é “colocá-los todos juntos” — interfaces. Dados podem ser perdidos através de uma interface; um componente pode ter um efeito inesperado ou adverso sobre outro, subfunções, quando combinadas, podem não produzir a função principal desejada; imprecisão aceitável individualmente pode ser amplificada em níveis não aceitáveis; estruturas de dados globais podem apresentar problemas. Infelizmente, a lista não acaba.

O teste de integração é uma técnica sistemática para construir a arquitetura de software ao mesmo tempo que conduz testes para descobrir erros associados com as interfaces. O objetivo é construir uma estrutura de programa determinada pelo projeto a partir de componentes testados em unidade.

Muitas vezes há uma tendência de tentar integração não incremental; isto é, construir o programa usando uma abordagem big bang. Todos os componentes são combinados com antecedência. O programa inteiro é testado como um todo. E, usualmente, o resultado é o caos!



Adotar a abordagem “big bang” é uma estratégia ineficaz, destinada a fracassar. Integre incrementalmente, testando enquanto trabalha.



Ao desenvolver um cronograma de projeto, você terá de considerar a maneira pela qual a integração ocorrerá, de forma que os componentes estejam disponíveis quando forem necessários.

São encontrados muitos erros. A correção é difícil porque o isolamento das causas é complicado pela vasta expansão do programa inteiro. Uma vez corrigidos esses erros, novos erros aparecem e o processo parece não ter fim.

A integração incremental é o oposto da abordagem big bang. O programa é construído e testado em pequenos incrementos, em que os erros são mais fáceis de isolar e corrigir; as interfaces têm maior probabilidade de ser testadas completamente; e uma abordagem sistemática de teste pode ser aplicada. Nos próximos parágrafos, algumas estratégias de integração incremental diferentes serão discutidas.

Integração descendente. *Teste de integração descendente (top-down)* é uma abordagem incremental para a construção da arquitetura de software. Os módulos são integrados deslocando-se para baixo através da hierarquia de controle, começando com o módulo de controle principal (programa principal). Módulos subordinados ao módulo de controle principal são incorporados à estrutura de uma maneira: primeiro-em-profundidade ou primeiro-em-largura (*depth-first* ou *breadth-first*).

Na Figura 17.5, *integração primeiro-em-profundidade* integra todos os componentes em um caminho de controle principal da estrutura do programa. A seleção de um caminho principal é de certa forma arbitrária e depende das características específicas da aplicação. Por exemplo, selecionando o caminho da esquerda, os componentes M_1 , M_2 , M_5 seriam integrados primeiro. Em seguida, M_8 ou (se necessário para o funcionamento apropriado de M_2) seria integrado M_6 . Depois, são criados os caminhos de controle central e da direita. A *integração primeiro-em-largura* incorpora todos os componentes diretamente subordinados a cada nível, movendo-se através da estrutura horizontalmente. Pela figura, os componentes M_2 , M_3 e M_4 seriam integrados primeiro. Em seguida vem o próximo nível de controle, M_5 , M_6 e assim por diante. O processo de integração é executado em uma série de cinco passos:

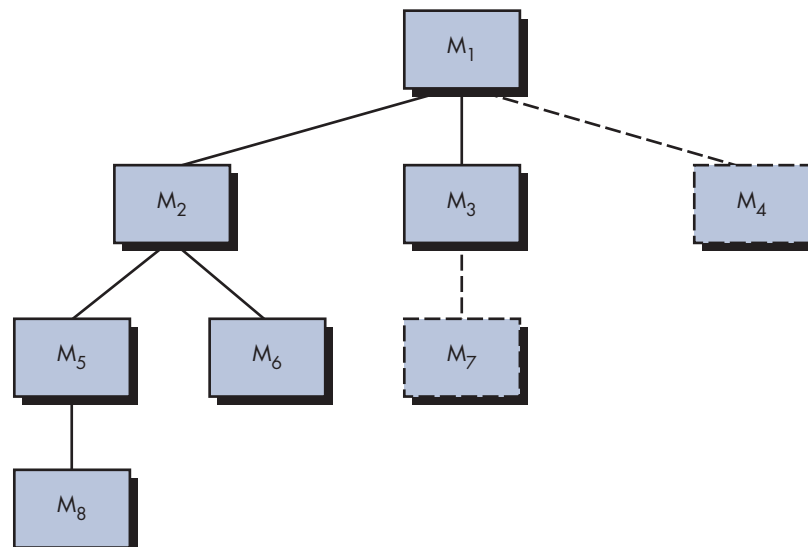


Quais são os passos para a integração descendente?

1. O módulo de controle principal é utilizado como um testador (*test driver*) e todos os componentes diretamente subordinados ao módulo de controle principal são substituídos por pseudocontroladores.
2. Dependendo da abordagem de integração selecionada (isto é, primeiro-em-profundidade ou primeiro-em-largura), pseudocontroladores subordinados são substituídos, um de cada vez, pelos componentes reais.

FIGURA 17.5

Integração descendente



3. Os testes são feitos à medida que cada componente é integrado.
4. Ao fim de cada conjunto de testes, outro pseudocontrolador substitui o componente real.
5. O teste de regressão (discutido mais adiante nesta seção) pode ser executado para garantir que não tenham sido introduzidos novos erros.

O processo continua a partir do passo 2 até que toda a estrutura do programa esteja concluída.

A estratégia de integração descendente verifica os principais pontos de controle ou decisão antecipada no processo de teste. Em uma estrutura de programa bem construída, a tomada de decisão ocorre nos níveis superiores na hierarquia e, portanto, é encontrada primeiro. Se existirem problemas de controle principal, um reconhecimento prévio é essencial. Se for selecionada a integração em profundidade, uma função completa do software pode ser implementada e demonstrada. A demonstração antecipada da capacidade funcional é um gerador de confiança para todos os interessados no programa.

A estratégia descendente parece relativamente descomplicada, mas na prática, podem surgir problemas logísticos. O mais comum desses problemas ocorre quando o processamento em baixos níveis na hierarquia é necessário para testar adequadamente níveis superiores. Pseudocontroladores substituem módulos de baixo nível no início do teste de cima para baixo; portanto, nenhum dado significativo pode fluir ascendentemente na estrutura do programa. Você, como testador, tem três escolhas: (1) adiar muitos testes até que os pseudocontroladores sejam substituídos pelos módulos reais, (2) desenvolver pseudocontroladores com funções limitadas que simulam o módulo real, ou (3) integrar o software de baixo para cima (ascendente).

A primeira abordagem (adiar os testes até que os pseudocontroladores sejam substituídos por módulos reais) pode fazer você perder algum controle sobre a correspondência entre testes específicos e incorporação de módulos específicos. Isso pode resultar em dificuldades para determinar a causa de erros e tende a violar a natureza altamente restrita da abordagem descendente. A segunda abordagem é prática, mas pode levar a uma complicação significativa, à medida que os pseudocontroladores se tornam mais e mais complexos. A terceira abordagem, chamada de integração ascendente, é discutida nos próximos parágrafos.

Integração ascendente. O teste de integração ascendente (*bottom-up*), como o nome diz, começa a construção e o teste com *módulos atômicos* (isto é, componentes nos níveis mais baixos na estrutura do programa). Devido aos componentes serem integrados de baixo para cima, a funcionalidade proporcionada por componentes subordinados a um dado nível está sempre disponível e a necessidade de pseudocontroladores é eliminada. Uma estratégia de integração ascendente pode ser implementada com os seguintes passos:

1. Componentes de baixo nível são combinados em agregados (*clusters*, também chamados de *builds*, construções) que executam uma subfunção específica de software.
2. Um pseudocontrolador (um programa de controle para teste) é escrito para coordenar entrada e saída do caso de teste.
3. O agregado é testado.
4. Os pseudocontroladores (*drivers*) são removidos, e os agregados são combinados movendo-se para cima na estrutura do programa.

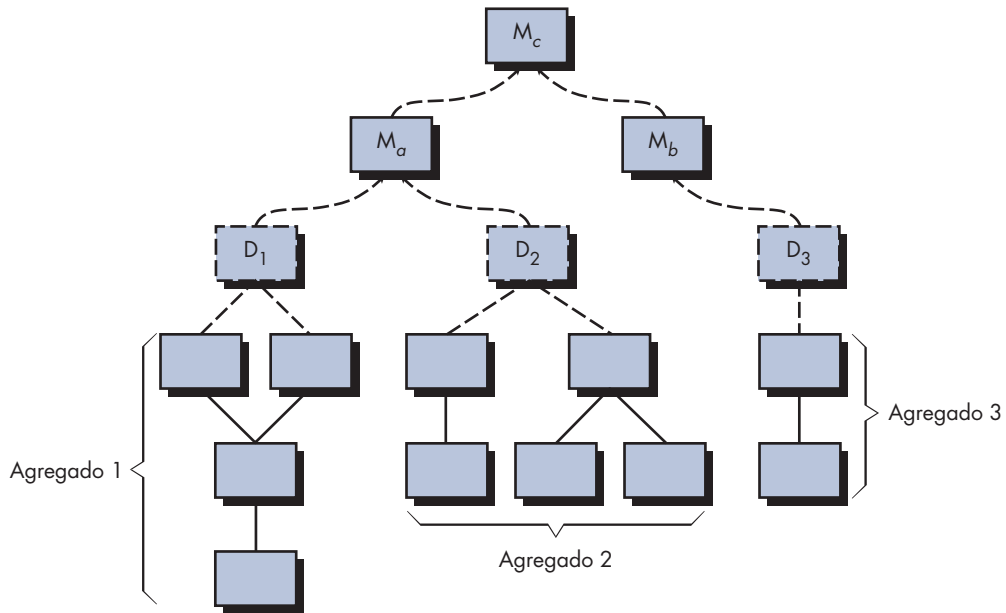
A integração segue o padrão ilustrado na Figura 17.6. Os componentes são combinados para formar os agregados (*clusters*) 1, 2 e 3. Cada um dos agregados é testado usando um pseudocontrolador (mostrado como um bloco tracejado). Componentes nos agregados 1 e 2 são subordinados a M_a . Os pseudocontroladores D_1 e D_2 são removidos e os agregados são interfaceados diretamente a M_a . De forma semelhante, o pseudocontrolador D_3 para o agregado 3 é removido antes da integração com o módulo M_b . M_a e M_b serão ambos integrados finalmente com o componente M_c , e assim por diante.

? Quais problemas podem ser encontrados quando é escolhida a integração descendente?

? Quais são os passos para a integração ascendente?

PONTO-CHAVE

A integração ascendente elimina a necessidade de pseudocontroladores complexos.

FIGURA 17.6**Integração
ascendente**

À medida que a integração se move para cima, a necessidade de pseudocontroladores de testes separados diminui. Na verdade, se os níveis descendentes da estrutura do programa forem integrados de cima para baixo, o número de pseudocontroladores pode ser bastante reduzido e a integração de agregados fica bastante simplificada.



O teste de regressão é uma estratégia importante para reduzir “efeitos colaterais”. Execute testes de regressão toda vez que for feita uma alteração grande no software (incluindo a integração de novos componentes).

Teste de regressão. Cada vez que um novo módulo é acrescentado como parte do teste de integração, o software muda. Novos caminhos de fluxo de dados são estabelecidos, podem ocorrer novas entradas e saídas e nova lógica de controle é chamada. Essas alterações podem causar problemas com funções que antes funcionavam corretamente. No contexto de uma estratégia de teste de integração, o teste de regressão é a reexecução do mesmo subconjunto de testes que já foram executados para assegurar que as alterações não tenham propagado efeitos colaterais indesejados.

Em um contexto mais amplo, testes bem-sucedidos (de qualquer tipo) resultam na descoberta de erros, e os erros devem ser corrigidos. Sempre que o software é corrigido, algum aspecto da configuração do software (o programa, sua documentação, ou os dados que o suportam) é alterado. O teste de regressão ajuda a garantir que as alterações (devido ao teste ou por outras razões) não introduzam comportamento indesejado ou erros adicionais.

O teste de regressão pode ser executado manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automáticas de captura/reexecução. Ferramentas de captura/reexecução permitem que o engenheiro de software capture casos de teste e resultados para reexecução e comparação subsequente. O conjunto de teste de regressão (o subconjunto de testes a ser executados) contém três classes diferentes de casos de teste:

- Uma amostra representativa dos testes que usam todas as funções do software.
- Testes adicionais que focalizam as funções de software que podem ser afetadas pela alteração.
- Testes que focalizam os componentes do software que foram alterados.

À medida que o teste de integração progride, o número de testes de regressão pode crescer muito. Portanto, o conjunto de testes de regressão deve ser projetado de forma a incluir somente aqueles testes que tratam de uma ou mais classes de erros em cada uma das funções principais do programa. É impraticável e ineficiente reexecutar todos os testes para todas as funções do programa quando ocorre uma alteração.

PONTO-CHAVE

O teste fumaça pode ser caracterizado como uma estratégia de integração rolante. O software é recriado (com novos componentes acrescentados) e o teste fumaça é realizado todos os dias.

Teste fumaça. *Teste fumaça* é uma abordagem de teste de integração usada frequentemente quando produtos de software são desenvolvidos. É projetado como um mecanismo de marcapasso para projetos com prazo crítico, permitindo que a equipe de software avalie o projeto frequentemente. Em essência, a abordagem teste fumaça abrange as seguintes atividades:

1. Componentes de software que foram traduzidos para um código são integrados em uma “construção” (*build*). Uma construção inclui todos os arquivos de dados, bibliotecas, módulos reutilizáveis e componentes necessários para implementar uma ou mais funções do produto.
2. Uma série de testes é criada para expor erros que impedem a construção de executar corretamente sua função. A finalidade deverá ser descobrir erros “bloqueadores” (*showstopper*) que apresentam a mais alta probabilidade de atrasar o cronograma do software.
3. A construção é integrada com outras construções, e o produto inteiro (em sua forma atual) passa diariamente pelo teste fumaça. A abordagem de integração pode ser descendente ou ascendente.

“Trate a construção diária (*daily build*) como o marcapasso do projeto. Se não houver marcapasso, o projeto está morto.”

Jim McCarthy

A frequência diária de teste do produto inteiro pode surpreender alguns leitores. No entanto, os testes frequentes dão, tanto aos gerentes quanto aos profissionais, uma ideia realística do progresso do teste de integração. McConnell [McC96] descreve o teste fumaça da seguinte maneira:

O teste fumaça deve usar o sistema inteiro de ponta a ponta. Ele não precisa ser exaustivo, mas deve ser capaz de expor os principais problemas. O teste fumaça deve ser bastante rigoroso de forma que, se a construção passar, você pode assumir que ele é estável o suficiente para ser testado mais rigorosamente.

O teste fumaça proporciona muitos benefícios quando aplicado a projetos de engenharia de software complexo e de prazo crítico:

- *O risco da integração é minimizado.* Devido aos teste fumaça serem feitos diariamente, as incompatibilidades e outros erros de bloqueio são descobertos logo, reduzindo assim a probabilidade de impacto sério no cronograma quando os erros são descobertos.
- *A qualidade do produto final é melhorada.* Devido ao fato de a abordagem ser orientada para a construção (integração), o teste fumaça pode descobrir erros funcionais, bem como erros de arquitetura e de projeto no nível de componente. Se esses erros forem corrigidos logo, resultará em melhor qualidade do produto.
- *O diagnóstico e a correção dos erros são simplificados.* Como todas as abordagens de teste de integração, os erros descobertos durante o teste fumaça provavelmente estarão associados com os “novos incrementos do software” — ou seja, o software que acaba de ser acrescentado à(s) construção(ões) é uma causa provável de um erro que acaba de ser descoberto.
- *É mais fácil avaliar o progresso.* A cada dia que passa, uma parte maior do software já está integrada e é demonstrado que funciona. Isso melhora o moral da equipe e dá aos gerentes uma boa indicação de que houve progressos.

Quais benefícios é possível obter do teste fumaça?

WebRef

Indicações para comentários sobre estratégias de teste podem ser encontradas em www.qalinks.com.

Opções estratégicas. Tem havido muita discussão (por exemplo, [Bei84]) sobre as vantagens e desvantagens relativas do teste de integração descendente *versus* ascendente. Em geral, as vantagens de uma estratégia tendem a resultar em desvantagens para outra. A maior desvantagem da abordagem descendente é a necessidade dos pseudocontroladores (*stubs*) e as dificuldades de teste que podem ser associadas com eles. Problemas associados com pseudocontroladores podem ser compensados pela vantagem de se testar antecipadamente as funções principais de controle. A maior desvantagem da integração ascendente é que “o programa como uma entidade não existe enquanto não for acrescentado o último módulo” [Mye79]. Essa desvantagem é compensada por um projeto de casos de teste mais simples e pela ausência de pseudocontroladores.

 O que é um “módulo crítico” e por que devemos identificá-lo?

A escolha de uma estratégia de integração depende das características do software e, algumas vezes, do cronograma do projeto. Em geral, uma abordagem combinada (também chamada de *teste sanduíche*), que usa testes descendentes para níveis superiores da estrutura do programa, acoplados com testes ascendentes para níveis subordinados, pode ser o melhor compromisso.

À medida que o teste de integração é conduzido, o testador deve identificar os módulos críticos. Um *módulo crítico* tem uma ou mais das seguintes características: (1) aborda vários requisitos de software, (2) tem um alto nível de controle (reside em posição relativamente alta na estrutura do programa), (3) é complexo ou sujeito a erros, ou (4) tem requisitos de desempenho bem definidos. Os módulos críticos devem ser testados o mais cedo possível. Além disso, os testes de regressão devem focalizar a função de módulo crítico.

Artefatos do teste de integração. Um plano global para integração do software e uma descrição dos testes específicos são documentados em uma *Especificação de Teste*. Esse produto incorpora um plano de teste e um documento de teste e torna-se parte da configuração do software. O teste é dividido em fases e construções que tratam de características funcionais e comportamentais específicas do software. Por exemplo, o teste de integração para o sistema de segurança *CasaSegura* pode ser dividido nas seguintes fases de teste:

- *Interação com o usuário* (entrada e saída de comandos, representação da tela, processamento e representação de erros)
- *Processamento do sensor* (aquisição da saída do sensor, determinação das condições do sensor, ações necessárias como consequência das condições)
- *Funções de comunicação* (habilidade para se comunicar com a estação de monitoramento central)
- *Processamento do alarme* (testes de ações do software que ocorrem quando um alarme é encontrado)

Cada uma dessas fases de teste de integração representa uma categoria funcional ampla dentro do software e geralmente pode ser relacionada a um domínio específico dentro da arquitetura do software. Portanto, são criadas construções de programa (grupos de módulos) para corresponder a cada fase. Os critérios e testes correspondentes a seguir são aplicados a todas as fases de teste:

 Qual critério deverá ser utilizado para projetar testes de integração?

Integridade da interface. As interfaces interna e externa são testadas à medida que cada módulo (ou agregado) é incorporado à estrutura.

Validade funcional. São executados testes destinados a descobrir erros funcionais.

Conteúdo de informação. São executados testes para descobrir erros associados com estruturas de dados locais ou globais.

Desempenho. São executados testes destinados a verificar os limites de desempenho estabelecidos durante o projeto do software.

Um cronograma para a integração, o desenvolvimento de software de uso geral e tópicos relacionados são também discutidos como parte do plano de teste. São estabelecidas as datas de início e fim para cada fase e são definidas “janelas de disponibilidade” para módulos submetidos a teste de unidade. Uma breve descrição do software de uso geral (pseudocontroladores e pseudocontrolados) concentra-se nas características que poderiam requerer dedicação especial. Finalmente, são descritos o ambiente e os recursos de teste. Configurações de hardware não usuais, simuladores exóticos e ferramentas ou técnicas especiais de teste são alguns dos muitos tópicos que também podem ser discutidos.

Em seguida, é descrito o procedimento detalhado de teste que é necessário para realizar o plano de teste. São descritas a ordem de integração e os testes correspondentes em cada etapa de integração. É incluída também uma lista de todos os casos de testes (anotados para referência subsequente) e dos resultados esperados.

Um histórico dos resultados reais do teste, problemas ou peculiaridades é registrado em um *Relatório de Teste* que pode ser anexado à *Especificação de Teste*, caso desejado. Essas informações podem ser vitais durante a manutenção do software. São apresentados também referências e apêndices apropriados.

Assim como quaisquer outros elementos de uma configuração de software, o formato da especificação de teste pode ser adaptado às necessidades locais de uma organização de engenharia de software. Porém, é importante notar que uma estratégia de integração (contida em um plano de teste) e detalhes do teste (descritos em um procedimento de teste) são ingredientes essenciais e devem estar presentes.

17.4 ESTRATÉGIAS DE TESTE PARA SOFTWARE ORIENTADO A OBJETO³

De forma simplificada, o objetivo do teste é encontrar o maior número possível de erros com um esforço gerenciável durante um intervalo de tempo realístico. Embora esse objetivo fundamental permaneça inalterado para software orientado a objetos, a natureza do software orientado a objeto muda tanto a estratégia quanto a tática de teste (Capítulo 19).

17.4.1 Teste de unidade no contexto OO

Quando consideramos o software orientado a objeto, o conceito de unidades se modifica. O encapsulamento controla a definição de classes e objetos. Isso significa que cada classe e cada instância de uma classe (objeto) empacotam atributos (dados) e as operações que manipulam esses dados. Uma classe encapsulada é usualmente o foco do teste de unidade. No entanto, operações (métodos) dentro da classe são as menores unidades testáveis. Como uma classe pode conter um conjunto de diferentes operações, e uma operação em particular pode existir como parte de um conjunto de diferentes classes, a tática aplicada a teste de unidade precisa modificar-se.

Não podemos mais testar uma única operação isoladamente (a visão convencional do teste de unidade) mas sim como parte de uma classe. Para ilustrar, considere uma hierarquia de classe na qual uma operação *X* é definida para a superclasse e é herdada por várias subclasses. Cada subclasse usa uma operação *X*, mas é aplicada dentro do contexto dos atributos e operações privadas definidas para a subclasse. Como o contexto no qual a operação *X* é utilizada varia de maneira sutil, torna-se necessário testar a operação *X* no contexto de cada uma das subclasses. Isso significa que testar a operação *X* isoladamente (a abordagem de teste de unidade convencional) é usualmente ineficaz no contexto orientado a objeto.

O teste de classe para software OO é o equivalente ao teste de unidade para software convencional. Diferentemente do teste de unidade do software convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem através da interface do módulo, o teste de classe para software OO é controlado pelas operações encapsuladas na classe e pelo estado de comportamento da classe.

17.4.2 Teste de integração no contexto OO

Devido ao software orientado a objeto não ter uma estrutura óbvia de controle hierárquico, as estratégias tradicionais de integração descendente e ascendente (Seção 17.3.2) têm pouco significado. Além disso, integrar operações uma de cada vez em uma classe (a abordagem convencional de integração incremental) frequentemente é impossível devido “às interações diretas e indiretas dos componentes que formam a classe” [Ber93].

Há duas estratégias diferentes para teste de integração de sistemas OO [Bin94b]. A primeira, *teste baseado em sequência de execução (thread-based testing)*, integra o conjunto de classes necessárias para responder a uma entrada ou um evento do sistema. Cada sequência de

PONTO-CHAVE

O teste de classe para OO é análogo ao teste de módulo para software convencional. Não é aconselhável testar operações isoladamente.

³ Os conceitos básicos orientados a objeto são apresentados no Apêndice 2.

PONTO-CHAVE

Uma estratégia importante para integração de software OO é o teste com base em sequência de execução (thread). Sequências de execução (threads) são conjuntos de classes que respondem a uma entrada ou evento. Testes fundamentados no uso focalizam classes que não colaboram intensamente com outras classes.

execução é integrada e testada individualmente. O teste de regressão é aplicado para garantir que não ocorram efeitos colaterais. A segunda abordagem de integração, *teste baseado em uso* (*use-based testing*), inicia a construção do sistema testando aquelas classes (chamadas de *classes independentes*) que usam poucas (ou nenhuma) classes *servidoras*. Depois que as classes independentes são testadas, o próximo nível de classes, chamadas de *classes dependentes*, que usam as classes independentes, são testadas. Essa sequência de camadas de teste de classes dependentes continua até que todo o sistema seja construído.

O uso de pseudocontroladores e pseudocontrolados também muda quando é executado o teste de integração de sistemas OO. Pseudocontroladores podem ser utilizados para testar operações no nível mais baixo e para o teste de grupos inteiros de classes. Um pseudocontrolador também pode ser usado para substituir a interface de usuário de maneira que os testes da funcionalidade do sistema podem ser conduzidos antes da implementação da interface. Pseudocontrolados podem ser usados em situações nas quais é necessária a colaboração entre classes, mas uma ou mais das classes colaboradoras ainda não foi totalmente implementada.

O *teste de agregado* (*cluster*) é uma etapa no teste de integração de software OO. Nesse caso, um agregado de classes colaboradoras (determinado examinando-se os modelos CRC e o modelo objeto-relacionamento) é exercitado projetando casos de teste que tentam descobrir erros nas colaborações.

17.5 ESTRATÉGIAS DE TESTE PARA WEBAPPS

A estratégia para teste de WebApp adota os princípios básicos para todo o teste de software e aplica táticas usadas em sistemas OO. Os seguintes passos resumem essa abordagem:

1. O modelo de conteúdo é revisado para descobrir erros.
2. O modelo de interface é revisado para assegurar que todos os casos podem ser acomodados.
3. O modelo de projeto da WebApp é revisado para descobrir erros de navegação.
4. A interface de usuário é testada para descobrir erros na mecânica de apresentação e/ou navegação.
5. Para cada componente funcional é feito o teste de unidade.
6. É testada a navegação através de toda a arquitetura.
7. A WebApp é implementada em uma variedade de ambientes com configurações diferentes e é testada quanto à compatibilidade com cada configuração.
8. São feitos testes de segurança tentando explorar vulnerabilidades na WebApp ou dentro de seu ambiente.
9. São feitos testes de desempenho.
10. A WebApp é testada por uma população de usuários finais controlados e monitorados. Os resultados da interação desses usuários com o sistema são avaliados quanto a erros de conteúdo e de navegação, problemas de utilização, problemas de compatibilidade e confiabilidade e desempenho da WebApp.

Devido a muitas WebApps evoluírem continuamente, o processo de teste é uma atividade contínua, conduzido pelo pessoal de suporte que usa testes de regressão derivados dos testes de desenvolvimento quando a WebApp foi desenvolvida inicialmente. Os métodos de teste para WebApp são considerados no Capítulo 20.

PONTO-CHAVE

A estratégia global para teste de WebApp pode ser resumida nos 10 passos descritos aqui.

WebRef

Excelentes artigos sobre teste de WebApp podem ser encontrados em www.stickyminds.com/testing.asp.

17.6 TESTE DE VALIDAÇÃO

O teste de validação começa quando termina o teste de integração, quando os componentes individuais já foram exercitados, o software está completamente montado como um pacote e os

**PONTO-
-CHAVE**

Como todas as outras etapas de teste, a validação tenta descobrir erros, mas o foco está no nível de requisitos — em coisas que ficarão imediatamente aparentes para o usuário final.

erros de interface já foram descobertos e corrigidos. No nível de validação ou de sistema, a distinção entre software convencional, software orientado a objeto e WebApps desaparece. O teste focaliza ações visíveis ao usuário e saídas do sistema reconhecíveis pelo usuário.

A validação pode ser definida de várias maneiras, mas uma definição simples (embora rigorosa) é que a validação tem sucesso quando o software funciona de uma maneira que pode ser razoavelmente esperada pelo cliente. Nesse ponto, um desenvolvedor de software veterano pode protestar: “Quem ou o quê é o árbitro para decidir o que são expectativas razoáveis?”. Se foi desenvolvida uma *Especificação de Requisitos de Software*, ela descreve todos os atributos do software visíveis ao usuário e contém uma seção denominada *Critério de Validação* que forma a base para uma abordagem de teste de validação.

17.6.1 Critério de teste de validação

A validação de software é conseguida por meio de uma série de testes que demonstram conformidade com os requisitos. Um plano de teste descreve as classes de testes a ser conduzidas e um procedimento de teste define casos de teste específicos destinados a garantir que todos os requisitos funcionais sejam satisfeitos, todas as características comportamentais sejam obtidas, todo o conteúdo seja preciso e adequadamente apresentado, todos os requisitos de desempenho sejam atendidos, a documentação esteja correta e outros requisitos sejam cumpridos (por exemplo, transportabilidade, compatibilidade, recuperação de erro, manutenibilidade).

Após cada caso de teste de validação ter sido conduzido, existe uma dentre duas possíveis condições: (1) a característica de função ou desempenho está de acordo com a especificação e é aceita ou (2) descobre-se um desvio da especificação e é criada uma lista de deficiências. Desvios ou erros descobertos nesse estágio de um projeto raramente podem ser corrigidos antes do prazo de entrega programado. Frequentemente, é necessário negociar com o cliente para estabelecer um método para resolver as deficiências.

17.6.2 Revisão da configuração

Um elemento importante do processo de validação é a *revisão de configuração*. A finalidade da revisão é garantir que todos os elementos da configuração do software tenham sido adequadamente desenvolvidos, estejam catalogados e tenham os detalhes necessários para amparar as atividades de suporte. A revisão de configuração, também chamada de auditoria, é discutida em mais detalhes no Capítulo 22.

17.6.3 Teste alfa e beta

É praticamente impossível para um desenvolvedor de software prever como o cliente realmente usará um programa. As instruções de uso podem ser mal interpretadas, combinações estranhas de dados podem ser usadas regularmente; resultados que pareciam claros para o testador podem ser confusos para um usuário no campo.

Quando é construído um software personalizado para um cliente, são feitos testes de aceitação para permitir ao cliente validar todos os requisitos. Conduzido pelo usuário final e não por engenheiros de software, um teste de aceitação pode variar de um informal *test drive* até uma série de testes planejados e sistematicamente executados. Na verdade, um teste de aceitação pode ser executado por um período de semanas ou meses, descobrindo assim erros cumulativos que poderiam degradar o sistema ao longo do tempo.

Se um software é desenvolvido como um produto para ser usado por muitos clientes, é impraticável executar testes formais de aceitação para cada cliente. Muitos construtores de software usam um processo chamado de teste alfa e beta para descobrir erros que somente o usuário final parece ser capaz de encontrar.

O *teste alfa* é conduzido na instalação do desenvolvedor por um grupo representativo de usuários finais. O software é usado em um cenário natural com o desenvolvedor “espiando por cima dos ombros” dos usuários, registrando os erros e os problemas de uso. Os testes alfa são conduzidos em um ambiente controlado.

“Com uma boa verificação, todos os erros ficam expostos (por exemplo, com uma base suficientemente ampla de testadores beta e codesenvolvedores, quase todos os problemas serão caracterizados rapidamente e a solução será óbvia para alguém).”

E. Raymond

? Qual é a diferença entre um teste alfa e um teste beta?

O teste *beta* é conduzido nas instalações de um ou mais usuários finais. Diferentemente do teste alfa, o desenvolvedor geralmente não está presente. Portanto, o teste beta é uma aplicação “ao vivo” do software em um ambiente que não pode ser controlado pelo desenvolvedor. O cliente registra todos os problemas (reais ou imaginários) encontrados durante o teste beta e relata esses problemas para o desenvolvedor em intervalos regulares. Como resultado dos problemas relatados durante o teste beta, os engenheiros de software fazem modificações e então preparam a liberação do software para todos os clientes.

Uma variação do teste beta, chamada de *teste de aceitação do cliente*, às vezes é executada quando é fornecido software personalizado a um cliente sob contrato. O cliente executa uma série de testes específicos na tentativa de descobrir erros antes de aceitar o software do desenvolvedor. Em alguns casos (por exemplo, um grande sistema corporativo ou governamental) o teste de aceitação pode ser muito formal e levar vários dias ou mesmo semanas.

CASA SEGURA



Preparando para validação

Cena: Escritório de Doug Miller, onde continua o projeto no nível de componente e a criação de certos componentes.

Participantes: Doug Miller, gerente de engenharia de software, Vinod, Jamie, Ed e Shakira — membros da equipe de engenharia de software do CasaSegura.

Conversa:

Doug: O primeiro incremento estará pronto para validação em... Vejamos, três semanas?

Vinod: Certo. A integração está indo bem. Estamos fazendo o teste fumaça diariamente, encontrando alguns erros, mas nada que não se possa resolver. Até agora, tudo bem.

Doug: Fale sobre a validação.

Shakira: Bem, usaremos como base para nosso projeto de teste todos os casos de uso. Ainda não comecei, mas desenvolveremos testes para todos os casos de uso pelos quais sou responsável.

Ed: A mesma coisa aqui.

Jamie: Eu também, mas temos de juntar as nossas ações para teste de aceitação e também para teste alfa e beta, não?

Doug: Sim. Na verdade, estive pensando que poderíamos contratar alguém para nos ajudar na validação. Tenho verba disponível no orçamento... E teríamos um novo ponto de vista.

Vinod: Eu acho que temos tudo sob controle.

Doug: Estou certo que sim, mas um ITG nos dá uma visão independente sobre o software.

Jamie: Estamos com o tempo apertado aqui, Doug. Não temos tempo suficiente para pajear qualquer um que você trouxer aqui.

Doug: Eu sei, eu sei. Mas se um ITG trabalhar a partir dos requisitos e casos de uso, não será necessário muito acompanhamento.

Vinod: Eu ainda acho que temos tudo sob controle.

Doug: Eu sei, Vinod, mas vou insistir nisso. Vamos marcar uma reunião com o representante do ITG ainda esta semana. Vamos dar o pontapé inicial e ver até onde eles chegam.

Vinod: Ok, talvez eles possam aliviar a carga.

17.7 TESTE DE SISTEMA

“Assim como a morte e os impostos, o teste é tanto desagradável quanto inevitável.”

Ed Yourdon

No início deste livro, chamamos a atenção para o fato de que o software é apenas um elemento de um grande sistema de computador. No final, o software é incorporado com outros elementos do sistema (por exemplo, hardware, pessoas, informações), e é executada uma série de testes de integração de sistema e validação. Esses testes estão fora do escopo do processo de software e não são executados somente por engenheiros de software. No entanto, as etapas executadas durante o projeto de software e o teste podem aumentar muito a probabilidade de uma integração de software bem-sucedida em um sistema maior.

Um problema clássico de teste de sistema é a “procura do culpado”. Isso ocorre quando é descoberto um erro e os desenvolvedores de diversos elementos do sistema começam a acusar um ao outro pelo problema. Em lugar de adotar essa postura sem sentido, você deve se antecipar aos problemas potenciais de interface e (1) criar caminhos de manipulação de erro que testem todas as informações vindas de outros elementos do sistema, (2) executar uma série de

testes que simulem dados incorretos ou outros erros potenciais na interface de software, (3) registrar os resultados dos testes para usar como “evidência” se ocorrer a caça ao culpado, e (4) participar do planejamento e projeto de testes do sistema para assegurar que o software seja testado adequadamente.

Teste de sistema é na realidade uma série de diferentes testes cuja finalidade primária é exercitar totalmente o sistema. Embora cada um dos testes tenha uma finalidade diferente, todos funcionam no sentido de verificar se os elementos do sistema foram integrados adequadamente e executam as funções a eles alocadas. Nas próximas seções, discutiremos os tipos de testes de sistema que são vantajosos para sistemas de software.

17.7.1 Teste de recuperação

Muitos sistemas de computador devem se recuperar de falhas e retomar o processamento em pouco ou nenhum tempo de parada. Em alguns casos, um sistema tem de ser tolerante a falhas; ou seja, falhas no processamento não devem causar a paralisação total do sistema. Em outros casos, uma falha no sistema deve ser corrigida dentro de um determinado período de tempo, caso contrário, poderão ocorrer sérios prejuízos financeiros.

Teste de recuperação é um teste do sistema que força o software a falhar de várias formas e verifica se a recuperação é executada corretamente. Se a recuperação for automática (executada pelo próprio sistema), a reinicialização, os mecanismos de verificação, recuperação de dados e reinício são avaliados quanto à correção. Se a recuperação requer intervenção humana, o tempo médio de reparo (*mean-time-to-repair* - MTTR) é avaliado para determinar se está dentro dos limites aceitáveis.

17.7.2 Teste de segurança

Qualquer sistema de computador que trabalhe com informações sensíveis ou que cause ações que podem inadequadamente prejudicar (ou beneficiar) indivíduos, é um alvo para acesso impróprio ou ilegal. As invasões abrangem uma ampla gama de atividades: hackers que tentam invadir sistemas por diversão, funcionários desgostosos que tentam invadir por vingança, indivíduos desonestos que tentam invadir para obter ganhos pessoais ilícitos.

O *teste de segurança* tenta verificar se os mecanismos de proteção incorporados ao sistema vão de fato protegê-lo contra acesso indevido. Citando Beizer [Bei84]: “A segurança de um sistema deve, naturalmente, ser testada quanto à invulnerabilidade por um ataque frontal — mas deve também ser testada quanto à invulnerabilidade por ataques laterais ou pela retaguarda”.

Durante o teste de segurança, o testador faz o papel do indivíduo que quer invadir o sistema. Vale tudo! O testador pode tentar obter senhas por meios externos; pode atacar o sistema com software personalizado projetado para romper quaisquer defesas que tenham sido criadas; pode sobrecarregar o sistema, o sistema pode assim recusar serviço a outros; pode causar erros no sistema proposadamente, esperando poder invadir durante a recuperação; pode examinar dados que não estão em segurança, tentando encontrar a chave para a entrada no sistema.

Com tempo e recursos suficientes, um bom teste de segurança finalmente conseguirá invadir o sistema. O papel do criador do sistema é tornar o custo da invasão maior do que o valor das informações que poderiam ser obtidas.

17.7.3 Teste por esforço

As etapas anteriores de teste resultaram em uma avaliação completa das funções normais do programa e do desempenho. Os testes por esforço (estresse) servem para colocar os programas em situações anormais. Essencialmente, o testador que executa teste por esforço pergunta: “Até onde podemos forçar o sistema até que ele falhe?”.

O *teste por esforço* usa um sistema de maneira que demande recursos em quantidade, frequência ou volumes anormais. Por exemplo, (1) testes especiais podem gerar dez interrupções por segundo, quando uma ou duas é a média normal, (2) a taxa de entrada de dados pode ser au-

“Se você está tentando encontrar verdadeiros defeitos no sistema e ainda não submeteu o seu software a um verdadeiro teste de esforço, então é hora de começar a fazê-lo.”

Boris Beizer

mentada de uma certa magnitude para determinar como as funções de entrada responderão, (3) são executados casos de teste que requerem o máximo de memória ou outros recursos, (4) são executados casos de teste que podem causar problemas de memória em um sistema operacional virtual, (5) são criados casos de teste que podem causar excessiva procura por dados residentes em disco. Essencialmente, o testador tenta quebrar o programa.

Uma variação do teste de esforço é uma técnica chamada *teste de sensibilidade*. Em algumas situações (as mais comuns ocorrem em algoritmos matemáticos), um intervalo muito pequeno de dados, contido dentro dos limites de dados válidos para um programa, podem causar um processamento extremo e até mesmo errôneo ou uma profunda degradação de desempenho. O teste de sensibilidade tenta descobrir combinações de dados dentro de classes de entrada válidas que podem causar instabilidade ou processamento inadequado.

17.7.4 Teste de desempenho

Para sistemas em tempo real e embutidos, um software que execute a função necessária mas não esteja em conformidade com os requisitos de desempenho é inaceitável. O teste de desempenho é projetado para testar o desempenho em tempo de execução do software dentro do contexto de um sistema integrado. O teste de desempenho é feito em todas as etapas no processo de teste. Até mesmo em nível de unidade, o desempenho de um módulo individual pode ser avaliado durante o teste. No entanto, o verdadeiro desempenho de um sistema só pode ser avaliado depois que todos os elementos do sistema estiverem totalmente integrados.

Os testes de desempenho muitas vezes são acoplados ao teste de esforço e usualmente requerem instrumentação de hardware e software. Isto é, frequentemente é necessário medir a utilização dos recursos (por exemplo, ciclos de processador) de forma precisa. Instrumentação externa pode monitorar intervalos de execução, *log* de eventos (por exemplo, interrupções) à medida que ocorrem, e verificar os estados da máquina regularmente. Monitorando o sistema com instrumentos, o testador pode descobrir situações que levam à degradação e possível falha do sistema.

17.7.5 Teste de disponibilização

Em muitos casos, o software deve operar em uma variedade de plataformas e sob mais de um ambiente de sistema operacional. O *teste de disponibilização*, também chamado de *teste de configuração*,

FERRAMENTAS DO SOFTWARE

Planejamento e Gerenciamento do Teste

Objetivo: Essas ferramentas ajudam a equipe de software no planejamento da estratégia de teste escolhida e a gerenciar o processo de teste enquanto ele é executado.



Mecanismos: as ferramentas nessa categoria cuidam do planejamento do teste, armazenamento do teste, gerenciamento e controle, rastreabilidade dos requisitos, integração, rastreamento de erro e geração de relatório. Os gerentes de projeto usam essas ferramentas para suplementar as ferramentas de cronograma de projeto. Testadores usam essas ferramentas para planejar atividades de teste e para controlar o fluxo de informações à medida que o processo de teste avança.

Ferramentas representativas:⁴

QaTraQ Test Case Management Tool, desenvolvida pela TraQ Software (www.testmanagement.com), “sugere uma abordagem estruturada para o gerenciamento do teste”.

QADirector, desenvolvida pela Compuware Corp. (www.compuware.com/qacenter), proporciona um ponto único de controle para gerenciamento de todas as fases do processo de teste.

TestWorks, desenvolvida pela Software Research, Inc. (www.soft.com/Products/index.html), contém um conjunto totalmente integrado de ferramentas de teste, incluindo ferramentas de gerenciamento de teste e relatórios.

OpensourceTesting.org (www.opensourcetesting.org/testmgt.php) lista uma variedade de ferramentas de gerenciamento e planejamento de teste open-source.

NI TestStand, desenvolvida pela National Instruments Corp. (www.ni.com), permite “desenvolver”, gerenciar, e executar sequências de teste escritas em qualquer linguagem de programação”.

⁴ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

exercita o software em cada ambiente no qual ele deve operar. Além disso, o teste de disponibilização examina todos os procedimentos de instalação e software especializado de instalação (por exemplo, os “instaladores”) que serão usados pelos clientes e toda a documentação que será usada para fornecer o software para os usuários finais.

Para citar um exemplo, considere a versão do software *CasaSegura* acessível pela Internet que permite ao cliente monitorar o sistema de segurança a partir de uma localização remota. A WebApp *CasaSegura* deve ser testada usando todos os navegadores para a Internet que podem ser encontrados. Um teste de disponibilização mais completo pode abranger combinações de navegadores com vários sistemas operacionais (por exemplo, Linux, Mac OS, Windows). Devido ao fato de a questão da segurança ser um problema importante, uma série completa de testes de segurança deveria ser integrada com o teste de disponibilização.

17.8 A ARTE DA DEPURAÇÃO

“Descobrimos, para nossa surpresa, que não era tão fácil obter os programas certos como pensávamos. Posso me lembrar do momento exato em que percebi que uma grande parte da minha vida desde então seria dedicada a encontrar os erros dos meus próprios programas.”

Maurice Wilkes
descobre a depuração, 1949



Procure evitar uma terceira consequência: A causa é encontrada, mas a “correção” não resolve o problema ou ainda introduz outro erro.



Porque a depuração é tão difícil?

O teste de software é um processo que pode ser sistematicamente planejado e especificado. Pode-se projetar casos de teste, uma estratégia pode ser definida e os resultados podem ser avaliados de acordo com as expectativas prescritas.

A *depuração* ocorre como consequência de um teste bem-sucedido. Isto é, quando um caso de teste descobre um erro, a depuração é o processo que resulta na remoção do erro. Embora a depuração possa e deva ser um processo ordenado, ela ainda é, em grande parte, uma arte. Um engenheiro de software, avaliando os resultados de um teste é frequentemente confrontado com uma indicação sintomática de um problema de software. Isso significa que a manifestação externa do erro e sua causa interna podem não ter nenhuma relação óbvia uma com a outra. O processo mental mal compreendido que conecta um sintoma a uma causa é a depuração.

17.8.1 O processo de depuração

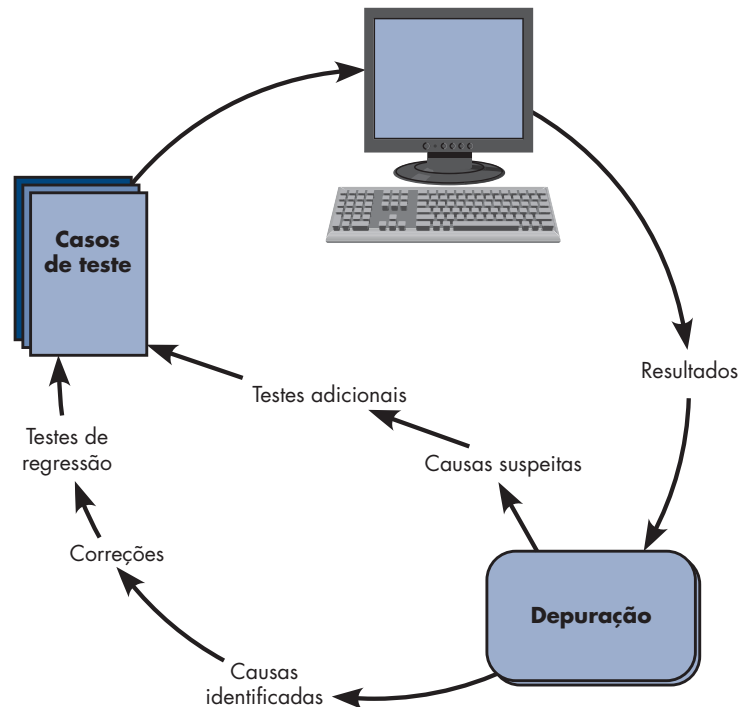
A depuração não é teste, mas frequentemente ocorre em consequência do teste.⁵ De acordo com a Figura 17.7, o processo de depuração começa com a execução de um caso de teste. Os resultados são avaliados e uma falta de correspondência entre o desempenho esperado e o desempenho real é encontrada. Em muitos casos, os dados não correspondentes são um sintoma de uma causa subjacente, embora oculta. O processo de depuração tenta combinar o sintoma com a causa, levando assim à correção do erro.

O processo de depuração usualmente apresentará um dentre dois resultados: (1) a causa será encontrada e corrigida ou (2) a causa não será encontrada. Neste último caso, quem está executando a depuração pode suspeitar de uma causa, criar um caso de teste para ajudar a confirmar aquela suspeita, e trabalhar na correção do erro de uma forma iterativa.

Porque a depuração é tão difícil? Com toda certeza, a psicologia humana (veja a Seção 17.8.2) tem muito mais a ver com a resposta do que a tecnologia de software. No entanto, algumas características dos erros fornecem alguns indícios:

1. O sintoma e a causa podem ser geograficamente remotos. Isto é, o sintoma pode aparecer em uma parte de um programa, enquanto a causa pode realmente estar localizada em um ponto muito afastado. Componentes altamente acoplados (Capítulo 8) pioram essa situação.
2. O sintoma pode desaparecer (temporariamente) quando outro erro for corrigido.
3. O sintoma pode ser, na realidade, causado por não erros (por exemplo, imprecisões de arredondamento).
4. O sintoma pode ser causado por erro humano que não é facilmente rastreável.
5. O sintoma pode ser um resultado de problemas de temporização, e não de problemas de processamento.

⁵ Ao fazer essa afirmativa, estamos assumindo a visão mais ampla possível do teste. Não só o desenvolvedor testa o software antes da entrega, mas também o cliente/usuário testa o software todas as vezes que ele é usado!

FIGURA 17.7**O processo de depuração**

"Todos sabem que a depuração é duas vezes mais difícil do que escrever um programa a partir de zero. Assim, se você foi o mais esperto possível ao escrever o programa, como poderá depurá-lo?"

Brian Kernighan

6. Pode ser difícil reproduzir com precisão as condições de entrada (por exemplo, uma aplicação em tempo real na qual a ordem das entradas é indeterminada).
7. O sintoma pode ser intermitente. Isso é particularmente comum em sistemas embutidos que acoplam hardware e software inextricavelmente.
8. O sintoma pode ocorrer devido a causas distribuídas por várias tarefas, executando em diferentes processadores.

Durante a depuração, você encontrará erros que variam desde levemente desagradáveis (por exemplo, um formato de saída incorreto) até catastróficos (por exemplo, o sistema falha, causando sérios danos econômicos ou físicos). À medida que as consequências de um erro aumentam, a pressão para encontrar o erro também aumenta. Às vezes, a pressão força alguns desenvolvedores de software a corrigir um erro e ao mesmo tempo introduzir outros dois.

17.8.2 Considerações psicológicas

Infelizmente, parece haver certa evidência de que a destreza na depuração é uma peculiaridade humana inata. Algumas pessoas são boas nisso e outras não. Embora a evidência experimental em depuração seja aberta a muitas interpretações, é possível observar uma grande variação na habilidade de depuração entre programadores com o mesmo nível de formação e experiência. Comentando os aspectos humanos da depuração, Shneiderman [Shn80] afirma:

Depuração é uma das partes mais frustrantes da programação. Ela tem elementos de solução de problemas ou quebra-cabeças, aliados à sensação desagradável de que você cometeu um erro. A ansiedade e indisposição para aceitar a possibilidade de erros aumentam a dificuldade da tarefa. Felizmente, há uma grande sensação de alívio e diminuição da tensão quando o erro finalmente é... Corrigido.

Embora possa ser difícil "aprender" a depurar, várias abordagens do problema podem ser propostas. Vamos examiná-las na Seção 17.8.3.

CASA SEGURA



Depuração

Cena: Cubículo de Ed enquanto é executado o teste de código e unidade.

Participantes: Ed e Shakira — membros da equipe de engenharia de software do CasaSegura.

Conversação:

Shakira (espiando pela entrada do cubículo): Olá... Onde você estava na hora do almoço?

Ed: Aqui mesmo... Trabalhando.

Shakira: Você parece tão mal... O que está havendo?

Ed (suspirando alto): Estive trabalhando neste... Erro desde que o descobri às 9:30 nesta manhã, e já são 2:45... E até agora, nada.

Shakira: Pensei que nós havíamos combinado que não devemos gastar mais de uma hora com problemas de depuração sozinhos; depois disso, procurar ajuda, certo?

Ed: Sim, mas...

Shakira (entrando no cubículo): Qual é o problema?

Ed: É complicado, e além disso, eu estive pesquisando por 5 horas. Você não vai poder resolvê-lo em 5 minutos.

Shakira: Desculpe-me... Qual é o problema?

[Ed explica o problema para Shakira, que o examina por 30 segundos sem dizer nada, então...]

Shakira (com um sorriso): Aha!, bem aqui, a variável denominada `setAlarmCondition`. Ela não deveria ser definida como "false" antes de iniciar o laço?

[Ed olha a tela incrédulo, inclina-se para frente e começa a bater com a cabeça no monitor. Shakira, rindo muito, se levanta e sai.]



Defina um limite, digamos, duas horas, para o tempo que você passa tentando depurar um problema por conta própria. Após esse tempo, procure ajuda!

17.8.3 Estratégias de depuração

Independentemente da abordagem adotada, a depuração tem um objetivo primordial — encontrar e corrigir a causa de um erro ou defeito de software. O objetivo é alcançado por uma combinação de avaliação sistemática, intuição e sorte. Bradley [Bra85] descreve a abordagem da depuração da seguinte maneira:

A depuração é uma aplicação direta do método científico que vem sendo desenvolvido há mais de 2.500 anos. A base da depuração é localizar a origem do problema [a causa] por particionamento binário, por meio de hipóteses que preveem novos valores a ser examinados.

Considere um exemplo simples que não é sobre software. Uma lâmpada na sua casa pode não acender. Se nada na casa funciona, a causa pode estar na chave geral da instalação elétrica ou na distribuição na rua; você observa os vizinhos e vê se eles também estão às escuras. Você pega a lâmpada suspeita e a conecta em outro soquete e pega alguma coisa que com certeza está funcionando e conecta naquela tomada suspeita. E assim continua a alternância entre hipótese e teste.

Em geral, foram propostas três estratégias de depuração [Mye79]: (1) força bruta, (2) rastreamento e (3) eliminação da causa. Cada uma dessas estratégias pode ser conduzida manualmente, mas as modernas ferramentas de depuração podem tornar o processo muito mais eficaz.

Táticas de depuração. A categoria *força bruta* para depuração é provavelmente o método mais comum e menos eficiente para isolar a causa de um erro de software. Usamos métodos de depuração do tipo força bruta quando tudo o mais falha. Usando uma filosofia do tipo "deixe o computador encontrar o erro", ocorrem despejos de memória, rastreamentos em tempo de execução e o programa é sobrecarregado com instruções de saída. Espera-se que no meio de toda aquela confusão de informações produzidas consigamos encontrar um indício que possa levar à causa de um erro. Embora a grande quantidade de informações produzidas possa no fim levar ao sucesso, mais frequentemente é uma perda de tempo e trabalho. É preciso pensar primeiro!

Rastreamento (backtracking) é uma abordagem comum de depuração que pode ser usada com sucesso em pequenos programas. Começando no ponto onde o sintoma foi descoberto, o código-fonte é investigado retroativamente (manualmente) até que a causa seja encontrada. Infelizmente, à medida que o número de linhas do código-fonte aumenta, o número de caminhos retroativos potenciais pode se tornar demasiadamente grande.

"O primeiro passo para reparar um programa com problema é fazê-lo falhar repetidamente (no exemplo mais simples possível)."

T. Duff

A terceira abordagem da depuração — *eliminação da causa* — é manifestada por indução ou dedução e introduz o conceito de particionamento binário. Os dados relacionados com a ocorrência do erro são organizados para isolar as causas potenciais. É imaginada uma “hipótese de causa” e os dados mencionados anteriormente são usados para provar ou negar a hipótese. Alternativamente, é preparada uma lista de todas as causas possíveis e são conduzidos os testes para eliminar cada uma delas. Se os testes iniciais indicam que uma determinada causa hipotética promete resultado, os dados são refinados tentando isolar o defeito.

Depuração automática. Cada uma dessas abordagens de depuração pode ser complementada com ferramentas de depuração que podem fornecer suporte semiautomático para o engenheiro de software à medida que são tentadas as estratégias. Hailpern e Santhanam [Hai02] resumem o estado dessas ferramentas quando afirmam “... muitas novas abordagens têm sido propostas e há muitos ambientes comerciais para depuração disponíveis. Ambientes Integrados de Desenvolvimento (IDE) proporcionam uma maneira de capturar alguns dos erros predeterminados específicos de linguagem (por exemplo, falta de caracteres de fim de instrução, variáveis indefinidas, e assim por diante) sem requerer compilação”. Há disponível uma ampla variedade de compiladores de depuração, auxílios dinâmicos de depuração (“rastreadores”), geradores de casos de teste automáticos e ferramentas de mapeamento de referências cruzadas. No entanto, as ferramentas não substituem uma avaliação cuidadosa fundamentada num modelo completo de projeto e um código-fonte claro.

FERRAMENTAS DO SOFTWARE



Depuração

Objetivo: essas ferramentas proporcionam auxílio automático para aqueles que devem depurar problemas de software. A intenção é fornecer informações que podem ser difíceis de obter se o processo de depuração for abordado manualmente.

Mecanismos: muitas ferramentas de depuração são específicas quanto à linguagem de programação e quanto ao ambiente.

Ferramentas representativas:⁶

Borland Gauntlet, distribuída pela Borland (www.borland.com), ajuda no teste e na depuração.

Coverty Prevent SQS, desenvolvida pela Coverty (www.coverty.com), proporciona auxílio de depuração para C++ e Java.

C++Test, desenvolvida pela Parasoft (www.parasoft.com), é uma ferramenta de teste de unidade que suporta uma ampla gama de testes em código C e C++. Características de depuração ajudam no diagnóstico de erros encontrados.

CodeMedic, desenvolvida pela NewPlanet Software (www.newplanetsoftware.com/medic/), proporciona uma interface gráfica para o depurador UNIX padrão, gdb, e implementa suas características mais importantes. O gdb atualmente suporta C/C++, Java, PalmOS, vários sistemas embutidos, linguagem Assembly, FORTRAN e Modula-2.

GNATS, um aplicativo freeware (www.gnu.org/software/gnats/), é um conjunto de ferramentas para rastrear bug reports.

O fator humano. Qualquer discussão sobre abordagens e ferramentas de depuração é incompleta se não mencionar um poderoso aliado — outras pessoas! Um ponto de vista novo, não ofuscado por horas de frustração, pode fazer maravilhas.⁷ Uma máxima final para a depuração seria: “Quando tudo mais falha, procure ajuda!”.

17.8.4 Correção do erro

Uma vez encontrado um defeito, ele precisa ser corrigido. Mas, conforme já observamos, a correção de um defeito pode introduzir outros erros e, portanto, causar mais danos do que

⁶ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nesta categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

⁷ O conceito de programação aos pares (recomendado como parte do modelo Extreme Programming discutido no Capítulo 3) proporciona um mecanismo para “depuração”, à medida que o software é projetado e codificado.

trazer benefícios. Van Vleck [Van89] sugere três perguntas simples que você deve fazer antes de fazer a “correção” que remove a causa de um defeito:

“O melhor testador não é aquele que encontra o maior número de erros... O melhor testador é aquele que consegue corrigir o maior número de erros.”

Cem Kaner et al.

1. *A causa do defeito é reproduzida em alguma outra parte do programa?* Em muitas situações, o defeito de um programa é causado por um padrão incorreto de lógica que pode ser reproduzido em qualquer outro ponto. Uma consideração explícita do padrão lógico pode resultar na descoberta de outros erros.
2. *Qual o “próximo defeito” que pode ser introduzido pelo reparo que estou prestes a fazer?* Antes de fazer a correção, o código-fonte (ou, melhor, o projeto) deverá ser avaliado para verificar acoplamento de lógica e estruturas de dados. Se a correção tem de ser feita em uma parte do programa altamente acoplada, deve ser tomado cuidado especial ao fazer qualquer alteração.
3. *O que poderíamos ter feito para evitar esse defeito já no início?* Essa pergunta é o primeiro passo para o estabelecimento de uma abordagem estatística de garantia de qualidade de software (Capítulo 16). Se você corrige o processo juntamente com o produto, o erro será removido do programa em questão e pode ser eliminado em todos os programas futuros.

17.9 RESUMO

O teste de software absorve a maior porcentagem do esforço técnico em um processo de software. Independentemente do tipo de software criado, uma estratégia para planejamento sistemático de teste, execução e controle começa considerando pequenos elementos do software e se encaminha para fora no sentido de abranger o programa como um todo.

O objetivo do teste de software é descobrir erros. Para o software convencional, esse objetivo é atingido por meio de uma série de passos de teste. Testes de unidade e de integração concentram-se na verificação funcional de um componente e na incorporação dos componentes na arquitetura de software. O teste de validação demonstra os requisitos de rastreabilidade do software, e o teste de sistema aprova o software quando ele é incorporado em um sistema maior. Cada passo do teste é realizado por meio de uma série de técnicas sistemáticas de teste que auxiliam no projeto dos casos de testes. Em cada passo do teste, o nível de abstração com o qual o software é considerado é ampliado.

A estratégia para teste de software OO começa com testes que exercitam as operações dentro de uma classe e depois passa para o teste baseado em sequências de execução para integração. Sequências de execução são conjuntos de classes que respondem a uma entrada ou evento. Testes baseados em uso focalizam classes que não colaboram intensamente com outras classes.

WebApps são testadas de maneira muito semelhante aos sistemas OO. No entanto, os testes são projetados para exercitar conteúdo, funcionalidade, interface, navegação e aspectos de desempenho da WebApp e segurança.

Diferentemente do teste (uma atividade sistemática, planejada), a depuração pode ser vista como uma arte. Começando com a indicação sintomática de um problema, a atividade de depuração deve rastrear a causa de um erro. Dentre os vários recursos disponíveis durante a depuração, o mais valioso é o conselho de outros membros da equipe de engenharia de software.

PROBLEMAS E PONTOS A PONDERAR

17.1. Usando as suas próprias palavras, descreva a diferença entre verificação e validação. Ambas usam métodos de projeto de caso de teste e estratégias de teste?

17.2. Liste alguns dos problemas que podem ser associados com a criação de um grupo de teste independente. Um grupo ITG e um grupo SQA são formados pelas mesmas pessoas?

17.3. É sempre possível desenvolver uma estratégia para teste de software que use a sequência de passos de teste descrita na Seção 17.1.3? Que possíveis complicações podem surgir para sistemas embutidos?

17.4. Porque um módulo altamente acoplado é difícil de testar em unidade?

17.5. O conceito de antidefeito (*antibugging*) (Seção 17.2.1) é uma maneira extremamente eficaz de proporcionar auxílio de depuração interno quando um erro é descoberto:

- a. Desenvolver uma série de diretrizes para antidepuração.
- b. Discutir vantagens do uso da técnica.
- c. Discutir as desvantagens.

17.6. Como o cronograma de projeto pode afetar o teste de integração?

17.7. O teste de unidade é possível ou até mesmo desejável em todas as circunstâncias? Forneça exemplos para justificar a sua resposta.

17.8. Quem deve executar o teste de validação — o desenvolvedor do software ou o usuário do software? Justifique a sua resposta.

17.9. Desenvolva uma estratégia de teste completa para o sistema *CasaSegura* discutido anteriormente neste livro. Documente-a em uma *Especificação de Teste*.

17.10. Como projeto de classe, desenvolva um *Guia de Depuração* para a sua instalação. O guia deve fornecer informações orientadas a linguagem e sistema que tenham sido aprendidas na escola de contratempos! Comece com um esboço dos tópicos que serão revisados pela classe e pelo seu professor. Distribua o guia para os outros no seu ambiente local.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Praticamente todos os livros sobre teste de software discutem estratégias juntamente com métodos para projeto de caso de teste. Everett e Raymond (*Software Testing*, Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing*, Wiley, 2007), Spiller e colaboradores (*Software Testing Process: Test Management*, Rocky Nook, 2007), Perry (*Effective Methods for Software Testing*, 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement*, 2d ed., Auerbach, 2004), Loveland et al. (*Software Testing Techniques*, Charles River Media, 2004), Burnstein (*Practical Software Testing*, Springer, 2003), Dustin (*Effective Software Testing*, Addison-Wesley, 2002), Craig and Kaskiel (*Systematic Software Testing*, Artech House, 2002), Tamres (*Introducing Software Testing*, Addison-Wesley, 2002), Whittaker (*How to Break Software*, Addison-Wesley, 2002) e Kaner et al. (*Lessons Learned in Software Testing*, Wiley, 2001) são apenas alguns dos muitos livros que discutem princípios, conceitos, estratégias e métodos de teste.

Para aqueles leitores que se interessam por métodos ágeis de desenvolvimento de software, Crispin e House (*Testing Extreme Programming*, Addison-Wesley, 2002) e Beck (*Test Driven Development: ByExample*, Addison-Wesley, 2002) apresentam estratégias de teste e táticas para Extreme Programming.

Kaner e colaboradores (*Lessons Learned in Software Testing*, Wiley, 2001) apresentam uma coleção de mais de 300 “lições” pragmáticas (diretrizes) que todo testador de software deveria conhecer.

Watkins (*Testing IT: An Off-the-Shelf Testing Process*, Cambridge University Press, 2001) estabelece uma estrutura de teste eficaz para todos os tipos de software desenvolvido e adquirido. Manges e O'Brien (*Agile Testing with Ruby and Rails*, Apress, 2008) tratam das estratégias de teste e técnicas para a linguagem de programação Ruby e estrutura para a Internet.

Sykes e McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir e Goel (*Testing Object-Oriented Software*, Springer-Verlag, 2000), Binder (*Testing Object-Oriented Systems*, Addison-Wesley, 1999), Kung et al. (*Testing Object-Oriented Software*,

IEEE Computer Society Press, 1998), e Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) apresentam estratégias e métodos para teste de sistemas orientados a objeto.

Diretrizes para depuração podem ser encontradas nos livros de Grötter et al. (*The Developer's Guide to Debugging*, Springer, 2008), Agans (*Debugging*, Amacon, 2006), Zeller (*Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann, 2005), Tells e Hsieh (*The Science of Debugging*, The Coreolis Group, 2001), e Robbins (*Debugging Applications*, Microsoft Press, 2000). Kaspersky (*Hacker Debugging Uncovered*, A-List Publishing, 2005) trata da tecnologia de ferramentas de depuração. Younessi (*Object-Oriented Defect Management of Software*, Prentice-Hall, 2002) apresenta técnicas para cuidar de defeitos encontrados em sistemas orientados a objeto.

Beizer [Bei84] apresenta uma interessante “sistemática de defeitos” que pode levar a métodos eficazes para planejamento de teste.

Livros dos autores Madisetti e Akgul (*Debugging Embedded Systems*, Springer, 2007), Robbins (*Debugging Microsoft .NET 2.0 Applications*, Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning*, Prentice-Hall, 2005), Ford e Teorey (*Practical Debugging in C++*, Prentice-Hall, 2002), Brown (*Debugging Perl*, McGraw-Hill, 2000), e Mitchell (*Debugging Java*, McGraw-Hill, 2000) tratam da natureza especial da depuração para os ambientes citados nos seus títulos.

Uma ampla gama de fontes de informação sobre métodos de teste orientado a objeto está disponível na Internet. Uma lista atualizada das referências relevantes para as técnicas de teste pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

18

TESTANDO APLICATIVOS CONVENCIONAIS

CONCEITOS-CHAVE

ambientes especializados...	446
análise de valor limite	442
complexidade ciclomática	433
grafos de fluxo...	432
matrizes gráficas .	436
métodos de teste baseados em grafos	440
padrões	449
particionamento de equivalência...	441
teste baseado em modelo	445
teste caixa-branca	431
teste caixa-preta.	439

O teste apresenta uma anomalia interessante para os engenheiros de software, que são, por natureza, pessoas construtivas. O teste requer que o desenvolvedor descarte noções preconcebidas da “corretividade” do software recém-desenvolvido e passe a trabalhar arduamente projetando casos de teste para “quebrar” o software. Beizer [Bei90] descreve essa situação eficazmente quando declara:

Há um mito de que, se fôssemos realmente bons em programação, não precisaríamos caçar erros. Se pudéssemos realmente nos concentrar, se todos usassem programação estruturada, projeto com detalhamento progressivo... Então não haveria erros. E assim continua o mito. Existem erros, diz o mito, porque somos ruins no que fazemos; e se somos ruins no que fazemos, devemos nos sentir culpados por isso. Portanto, o teste e o planejamento de casos de teste é um reconhecimento de falha, que sugere uma boa dose de culpa. E o tédio do teste é exatamente a punição pelos nossos erros. Punição por quê? Por sermos humanos? Culpados de quê? De não conseguir atingir a perfeição desumana? De não distinguir entre o que um outro programador pensa e o que ele diz? Por não conseguir ser telepático? Por não resolver problemas de comunicação humana que já existem... Há quarenta séculos?

O teste deve realmente insinuar culpa? O teste é realmente destrutivo? A resposta a essas questões é “Não!”

PANORAMA

O que é? Uma vez gerado o código-fonte, o software deve ser testado para descobrir (e corrigir) tantos erros quanto possível antes de fornecê-lo ao seu cliente. Sua meta é projetar um conjunto de casos de teste que tenha a mais alta probabilidade de encontrar erros — mas como? É aqui que entram em cena as técnicas de teste de software. Essas técnicas fornecem diretrizes sistemáticas para projetar testes que (1) exercitam a lógica interna e as interfaces de todos os componentes do software e (2) exercitam os domínios de entrada e saída do programa para descobrir erros no funcionamento, comportamento e desempenho do programa.

Quem realiza? Durante os primeiros estágios do teste, um engenheiro de software executa todos os testes. Porém, à medida que o processo de teste avança, especialistas podem ser envolvidos.

Por que é importante? Revisões e outras ações SQA podem descobrir e realmente descobrem erros, mas não são suficientes. Toda vez que o programa é executado, o cliente testa-o! Portanto, você tem de executar o programa antes que ele chegue ao cliente com o objetivo específico de encontrar e remover todos os erros. Para encontrar o maior número possível de erros, devem ser executados testes sistematicamente, e os casos de teste devem ser projetados usando técnicas disciplinadas.

Quais são as etapas envolvidas? Para aplicações convencionais, o software é testado a partir de duas perspectivas diferentes: (1) a lógica interna do programa é exercitada usando técnicas de projeto de caso de teste “caixa branca” e (2) os requisitos de software são exercitados usando técnicas de projeto de casos de teste “caixa preta”. Casos de teste de uso ajudam no projeto de testes para descobrir erros no nível de validação de software. Em todos os casos, a intenção é encontrar o número máximo de erros com o mínimo de esforço e tempo.

Qual é o artefato? Um conjunto de casos de teste projetados para exercitar a lógica interna, interfaces, colaborações entre componentes e os requisitos externos é projetado e documentado, os resultados esperados são definidos e os resultados obtidos são registrados.

Como garantir que o trabalho foi feito corretamente? Quando você começar o teste, mude o seu ponto de vista. Tente “quebrar” o software! Projete casos de teste de forma disciplinada e reveja os casos de teste que você criou, quanto à perfeição. Além disso, você pode avaliar a abrangência do teste e monitorar as atividades de detecção de erros.

teste de estrutura de controle	437
teste do caminho básico.	485
teste de matriz ortogonal	442

Neste capítulo, discutimos técnicas para projetar casos de teste de software para aplicações convencionais. O projeto de casos de teste focaliza um conjunto de técnicas para a criação de casos de teste, que satisfazem os objetivos globais e as estratégias de teste discutidas no Capítulo 17.

18.1 FUNDAMENTOS DO TESTE DE SOFTWARE

“Todo programa faz alguma coisa certa, só que pode não ser aquilo que queremos que ele faça.”

Autor desconhecido

? Quais são as características da testabilidade?

“Erros são mais comuns, mais disseminados e mais problemáticos no software do que em outras tecnologias.”

David Parnas

O objetivo do teste é encontrar erros, e um bom teste é aquele que tem alta probabilidade de encontrar um erro. Portanto, um engenheiro de software deve projetar e implementar um sistema ou produto baseado em computador tendo em mente a “testabilidade”. Ao mesmo tempo, os próprios testes devem ter uma série de características que permitam atingir o objetivo de encontrar o maior número de erros com o mínimo de esforço.

Testabilidade. James Bach¹ dá a seguinte definição para testabilidade: “*Testabilidade de software* é simplesmente a facilidade com que um programa de computador pode ser testado”. As seguintes características levam a um software testável:

Operabilidade. “Quanto melhor funcionar, mais eficientemente pode ser testado.” Se um sistema for projetado e implementado tendo em mente a qualidade, haverá poucos defeitos bloqueando a execução dos testes, permitindo que o teste ocorra sem sobressaltos.

Observabilidade. “O que você vê é o que você testa.” Entradas fornecidas como parte do teste produzem saídas distintas. Estados e variáveis do sistema são visíveis ou podem ser consultados durante a execução. Saída incorreta é facilmente identificada. Erros internos são automaticamente detectados e relatados. O código-fonte é acessível.

Controlabilidade. “Quanto melhor pudermos controlar o software, mais o teste pode ser automatizado e otimizado.” Todas as possíveis saídas podem ser geradas por meio de alguma combinação de entrada, e os formatos de entrada e saída são consistentes e estruturados. Todo o código é executável através de alguma combinação de entrada. Estados e variáveis de software e hardware podem ser controlados diretamente pelo engenheiro de teste. Os testes podem ser convenientemente especificados, automatizados e reproduzidos.

Decomponibilidade. “Controlando o escopo do teste, podemos isolar problemas mais rapidamente e executar um reteste mais racionalmente.” O sistema de software é construído a partir de módulos independentes que podem ser testados de forma independente.

Simplicidade. “Quanto menos tiver que testar, mais rapidamente podemos testá-lo.” O programa deverá ter *simplicidade funcional* (por exemplo, o conjunto de características é o mínimo necessário para satisfazer os requisitos); *simplicidade estrutural* (por exemplo, a arquitetura é modularizada para limitar a propagação de falhas), e *simplicidade de código* (por exemplo, é adotado um padrão de codificação para facilitar a inspeção e a manutenção).

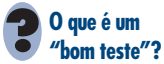
Estabilidade. “Quanto menos alterações, menos interrupções no teste.” As alterações no software são pouco frequentes, controladas quando elas ocorrem e não invalidam os testes existentes. O software recupera-se bem das falhas.

Compreensibilidade. “Quanto mais informações tivermos, mais inteligente será o teste.” O projeto arquitetural e as dependências entre componentes internos, externos e compartilhados são bem compreendidas. A documentação técnica é instantaneamente acessível, bem organizada, específica, detalhada e precisa. Alterações no projeto são comunicadas aos testadores.

¹ Os parágrafos seguintes são usados com permissão de James Bach (copyright 1994) e foram adaptados a partir de material que originalmente apareceu em uma postagem no newsgroup comp.software-eng.

Os atributos sugeridos por Bach podem ser utilizados por um engenheiro de software para desenvolver uma configuração de software (isto é, programas, dados e documentos) que é sensível ao teste.

Características do teste. E quanto aos próprios testes? Kaner, Falk e Nguyen [Kan93] sugerem os seguintes atributos para um “bom” teste:



O que é um “bom teste”?

Um bom teste tem alta probabilidade de encontrar um erro. Para atingir esse objetivo, o testador deve entender o software e tentar desenvolver uma imagem mental de como o software pode falhar. O ideal é que as classes de falhas sejam investigadas. Por exemplo, uma classe de falha em potencial, em uma interface gráfica com o usuário é uma falha em reconhecer a posição correta do mouse. Seria preparado um conjunto de testes para exercitar o mouse na tentativa de demonstrar erro no reconhecimento da posição do mouse.

Um bom teste não é redundante. O tempo e os recursos de teste são limitados. Não tem sentido realizar um teste que tenha a mesma finalidade de outro teste. Cada teste deve ter uma finalidade diferente (mesmo que seja sutilmente diferente).

Um bom teste deverá ser “o melhor da raça” [Kan93]. Em um grupo de testes com finalidades similares, as limitações de tempo e recursos podem induzir à execução de apenas um subconjunto desses testes. Nesses casos, deverá ser usado o teste que tenha a maior probabilidade de revelar uma classe inteira de erros.

Um bom teste não deve ser nem muito simples nem muito complexo. Embora seja possível combinar algumas vezes uma série de testes em um caso de teste, os possíveis efeitos colaterais associados com essa abordagem podem mascarar erros. Em geral, cada teste deve ser executado separadamente.

CASA SEGURA



Projetando testes únicos

Cena: Sala de Vinod.

Participantes: Vinod e Ed — membros da equipe de engenharia de software do CasaSegura.

Conversa:

Vinod: Então, esses são os casos de teste que você pretende usar para a operação `validacaoDeSenha`.

Ed: Sim, eles devem abranger muito bem todas as possibilidades para os tipos de senhas que um usuário possa introduzir.

Vinod: Vejamos... Você observou que a senha correta será 8080, certo?

Ed: Hã, hã.

Vinod: E você especifica as senhas 1234 e 6789 para testar o erro no reconhecimento de senhas inválidas?

Ed: Certo, e eu também testo senhas que são semelhantes à senha correta, veja... 8081 e 8180.

Vinod: Estas parecem OK, mas eu não vejo muito sentido em testar 1234 e 6789. Elas são redundantes... Testam a mesma coisa, não é isso?

Ed: Bem, são valores diferentes.

Vinod: É verdade, mas se 1234 não descobrir um erro... Em outras palavras, se a operação `validacaoDeSenha` percebe que é uma senha inválida, não é provável que 6789 nos mostre algo novo.

Ed: Entendo o que você quer dizer.

Vinod: Não estou tentando ser chato aqui... É que nós temos um tempo limitado para testar, portanto é uma boa ideia executar testes que tenham alta possibilidade de encontrar novos erros.

Ed: Sem problemas... Vou pensar um pouco mais nisso.

18.2 VISÕES INTERNA E EXTERNA DO TESTE

Qualquer produto de engenharia (e muitas outras coisas) pode ser testado por uma de duas maneiras: (1) Conhecendo a função especificada para o qual um produto foi projetado para realizar, podem ser feitos testes que demonstram que cada uma das funções é totalmente operacional, embora ao mesmo tempo procurem erros em cada função. (2) Conhecendo o funciona-

"Há apenas uma regra no projeto de casos de teste: abranger todas as características; mas não faça muitos casos de teste."

Tsuneo Yamaura

PONTO-CHAVE

Testes caixa-branca só podem ser projetados depois que o projeto no nível de componente (ou código-fonte) existir. Os detalhes lógicos do programa devem estar disponíveis.

mento interno de um produto, podem ser realizados testes para garantir que "tudo se encaixa", isto é, que as operações internas foram realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados. A primeira abordagem de teste usa uma visão externa e é chamada de teste caixa-preta. A segunda abordagem requer uma visão interna e é chamada de teste caixa-branca.²

O teste *caixa-preta* faz referência a testes realizados na interface do software. Um teste caixa-preta examina alguns aspectos fundamentais de um sistema, com pouca preocupação em relação à estrutura lógica interna do software. O teste *caixa-branca* fundamenta-se em um exame rigoroso do detalhe procedimental. Os caminhos lógicos do software e as colaborações entre componentes são testados exercitando conjuntos específicos de condições e/ou ciclos.

À primeira vista poderia parecer que um teste caixa-branca realizado de forma rigorosa resultaria em "programas 100% corretos". Tudo o que seria preciso fazer seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, ou seja, gerar casos de teste para exercitar a lógica do programa de forma exaustiva. Infelizmente, o teste exaustivo apresenta certos problemas logísticos. Mesmo para pequenos programas, o número de caminhos lógicos possíveis pode ser muito grande. No entanto, o teste caixa-branca não deve ser descartado como impraticável. Um número limitado de caminhos lógicos importantes pode ser selecionado e exercitado. Estruturas de dados importantes podem ser investigadas quando à validade.

INFORMAÇÕES



Teste Exaustivo

Considere um programa de 100 linhas em linguagem C. Após algumas declarações básicas de dados, o programa contém dois laços aninhados que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo, são necessárias 4 construções if-then-else. Há aproximadamente 10^{14} caminhos possíveis que podem ser executados nesse programa!

Para colocar esse número sob perspectiva, vamos supor que um processador de teste mágico ("mágico" porque não

existe tal processador) tenha sido desenvolvido para teste exaustivo. O processador pode desenvolver um caso de teste, executá-lo e avaliar os resultados em um milissegundo. Trabalhando 24 horas por dia, 365 dias por ano, o processador gastaria 3.170 anos para testar o programa. Isso, sem dúvida, tumultuaria qualquer cronograma de desenvolvimento.

Portanto, é razoável afirmar que o teste exaustivo é impossível para grandes sistemas de software.

18.3 TESTE CAIXA-BRANCA

"Os defeitos ficam à espreita nas esquinas e se reúnem nas fronteiras."

Boris Beizer

O teste *caixa-branca*, também chamado de teste *da caixa-de-vidro*, é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de software pode criar casos de teste que (1) garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez, (2) exercitam todas as decisões lógicas nos seus estados verdadeiro e falso, (3) executam todos os ciclos em seus limites e dentro de suas fronteiras operacionais, e (4) exercitam estruturas de dados internas para assegurar a sua validade.

18.4 TESTE DO CAMINHO BÁSICO

O teste *de caminho básico* é uma técnica de teste caixa-branca proposta por Tom McCabe [McC76]. O teste de caminho básico permite ao projetista de casos de teste derivar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para

² Os termos *teste funcional* e *teste estrutural* são às vezes usados em lugar de teste caixa-preta e teste caixa-branca, respectivamente.

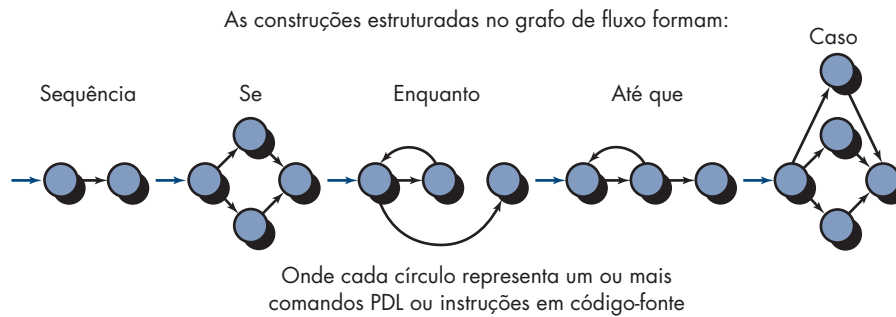
definir um conjunto base de caminhos de execução. Casos de teste criados para exercitar o conjunto básico executam com certeza todas as instruções de um programa pelo menos uma vez durante o teste.

18.4.1 Notação de grafo de fluxo

Antes de considerarmos o teste de caminho básico, deve ser introduzida uma notação simples para a representação do fluxo de controle, chamada de *grafo de fluxo* (ou *grafo de programa*).³ O grafo de fluxo representa o fluxo de controle lógico usando a notação ilustrada na Figura 18.1. Cada construção estruturada (Capítulo 10) tem um símbolo correspondente no grafo de fluxo.

FIGURA 18.1

Notação de grafo de fluxo

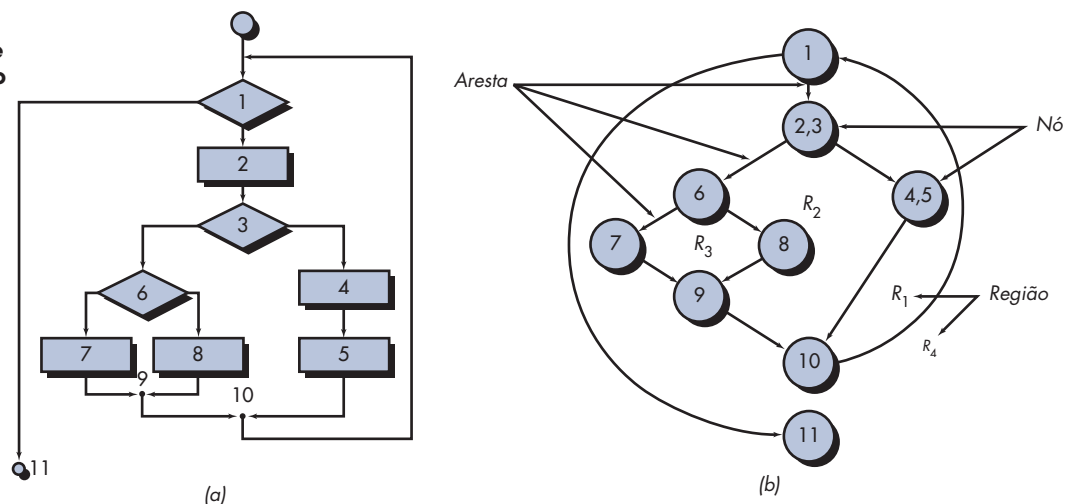


Um grafo de fluxo somente deve ser desenhado quando a estrutura lógica de um componente for complexa. O grafo de fluxo permite seguir os caminhos de programa mais facilmente.

Para ilustrar o uso de um grafo de fluxo, considere a representação do projeto procedimental na Figura 18.2a. É usado um fluxograma para mostrar a estrutura de controle do programa. A Figura 18.2b mapeia o fluxograma em um grafo de fluxo correspondente (considerando que os losangos de decisão do fluxograma não contêm nenhuma condição composta). Na Figura 18.2b, cada círculo, chamado de *nó do grafo de fluxo*, representa um ou mais comandos procedurais. Uma sequência de retângulos de processamento e um losango de decisão podem ser mapeados em um único nó. As setas no grafo de fluxo, chamadas de *arestas* ou *ligações*, representam fluxo de controle e são análogas às setas do fluxograma. Uma aresta deve terminar em um nó, mesmo que esse nó não represente qualquer comando procedural (por exemplo, veja o símbolo do diagrama de fluxo para a construção se-então-senão - if-then-else). As áreas limitadas por arestas e nós são chamadas de *regiões*. Ao contarmos as regiões, incluímos a área fora do grafo como uma região.⁴

FIGURA 18.2

(a) Fluxograma e (b) grafo de fluxo

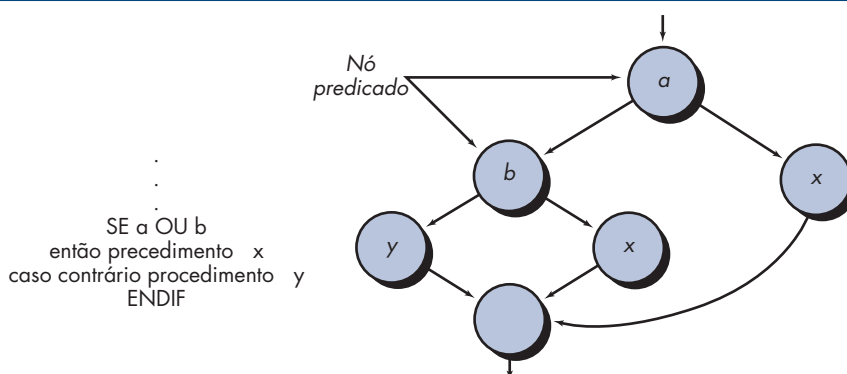


³ Na realidade, o método do caminho básico pode ser executado sem o uso de grafos de fluxo. No entanto, eles servem como uma notação útil para entender o fluxo de controle e ilustrar a abordagem.

⁴ Uma discussão mais detalhada sobre grafos e seus usos é apresentada na Seção 18.6.1.

FIGURA 18.3

Lógica composta



Quando condições compostas são encontradas em um projeto procedimental, a geração de um grafo de fluxo torna-se ligeiramente mais complicada. Uma condição composta ocorre quando um ou mais operadores booleanos (OR, AND, NAND, NOR lógicos) estão presentes em um comando condicional. De acordo com a Figura 18.3, o trecho de PDL (*program design language*) é traduzido no grafo de fluxo mostrado. Note que é criado um nó separado para cada uma das condições *a* e *b* no comando SE *a* OU *b*. Cada nó contendo uma condição é chamado de *nó predicado* (*predicate node*) e é caracterizado por duas ou mais arestas saindo dele.

18.4.2 Caminhos de programa independentes

Um *caminho independente* é qualquer caminho através do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando definido em termos de um grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes de o caminho ser definido. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo ilustrado na Figura 18.2b é

Caminho 1: 1-11

Caminho 2: 1-2-3-4-5-10-1-11

Caminho 3: 1-2-3-6-8-9-10-1-11

Caminho 4: 1-2-3-6-7-9-10-1-11

Note que cada novo caminho introduz uma nova aresta. O caminho

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

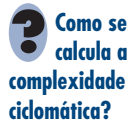
não é considerado um caminho independente porque ele é simplesmente uma combinação dos caminhos já especificados e não atravessa nenhuma nova aresta.

Os caminhos de 1 a 4 constituem um *conjunto base* para o grafo de fluxo da Figura 18.2b. Isto é, se testes podem ser projetados para forçar a execução desses caminhos (conjunto base), cada comando do programa terá sido executado com certeza pelo menos uma vez e cada condição terá sido executada em seus lados verdadeiro e falso. Deve-se notar que o conjunto base não é único. De fato, vários conjuntos base diferentes podem ser derivados para um dado projeto procedimental.

Como sabemos quantos caminhos procurar? O cálculo da complexidade ciclomática fornece a resposta. *Complexidade ciclomática* é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do método de teste de caminho básico, o valor computado para a complexidade ciclomática define o número de caminhos independentes no conjunto base de um programa, fornecendo um limite superior para a quantidade de testes que devem ser realizados para garantir que todos os comandos tenham sido executados pelo menos uma vez.



A complexidade ciclomática é uma métrica útil para previsão dos módulos que têm a tendência de apresentar erros. Ela pode ser usada tanto para o planejamento de teste quanto para projeto de casos de teste.



Como se calcula a complexidade ciclomática?

A complexidade ciclomática tem um fundamento na teoria dos grafos e fornece uma métrica de software extremamente útil. A complexidade é calculada por uma de três maneiras:

1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática.
2. A complexidade ciclomática $V(G)$ para um grafo de fluxo G é definida como

$$V(G) = E - N + 2$$
 em que E é o número de arestas do grafo de fluxo e N é o número de nós do grafo de fluxo.
3. A complexidade ciclomática $V(G)$ para um grafo de fluxo G também é definida como

$$V(G) = P + 1$$
 em que P é o número de nós predicados contidos no grafo de fluxo G .

PONTO-CHAVE

A complexidade ciclomática fornece o limite superior no número de casos de teste que precisam ser executados para garantir que cada comando do programa tenha sido executado pelo menos uma vez.

Examinando mais uma vez o diagrama de fluxo da Figura 18.2b, a complexidade ciclomática pode ser calculada usando cada um dos algoritmos citados anteriormente:

1. O grafo de fluxo tem quatro regiões.
2. $V(G) = 11$ arestas $- 9$ nós $+ 2 = 4$.
3. $V(G) = 3$ nós predicados $+ 1 = 4$.

Portanto, a complexidade ciclomática para o grafo de fluxo da Figura 18.2b é 4.

E o mais importante, o valor para $V(G)$ fornece um limite superior para o número de caminhos independentes que podem formar o conjunto base e, como consequência, um limite superior sobre o número de testes que devem ser projetados e executados para garantir a abrangência de todos os comandos do programa.

CASA SEGURA



Usando a complexidade ciclomática

Cena: Sala da Shakira.

Participantes: Vinod e Shakira — mem-

bros da equipe de engenharia de software do CasaSegura que estão trabalhando no planejamento de teste para as funções de segurança.

Conversa:

Shakira: Olha... Eu sei que deveríamos fazer o teste de unidade em todos os componentes da função de segurança, mas eles são muitos, e se você considerar o número de operações que precisam ser exercitadas, eu não sei... Talvez devêssemos esquecer o teste caixa-branca, integrar tudo e começar a aplicar os testes caixa-preta.

Vinod: Você acha que não temos tempo suficiente para fazer o teste dos componentes, realizar as operações e então integrar?

Shakira: O prazo final para o primeiro incremento está se esgotando e eu gostaria de... Oh, estou preocupada.

Vinod: Porque você não aplica testes caixa-branca pelo menos nas operações que têm maior probabilidade de apresentar erros?

Shakira (desesperada): E como eu posso saber exatamente quais são as que têm maior possibilidade de erro?

Vinod: V de G .

Shakira: Hã?

Vinod: Complexidade ciclomática — V de G . Basta calcular $V(G)$ para cada uma das operações dentro de cada um dos componentes e ver quais têm os maiores valores para $V(G)$. São essas que têm maior tendência a apresentar erro.

Shakira: E como eu calculo V de G ?

Vinod: É muito fácil. Aqui está um livro que descreve como fazer.

Shakira (folheando o livro): OK, não parece difícil. Vou tentar. As operações que tiverem os maiores $V(G)$ serão as candidatas aos testes caixa-branca.

Vinod: Mas lembre-se de que não há nenhuma garantia. Um componente com baixo valor $V(G)$ pode ainda ser sensível a erro.

Shakira: Tudo bem. Isso pelo menos me ajuda a limitar o número de componentes que precisam passar pelo teste caixa-branca.

"O foguete Ariane 5 explodiu no lançamento simplesmente por um defeito de software (uma falha) envolvendo a conversão de um valor em ponto flutuante de 64 bits em um inteiro de 16 bits. O foguete e seus quatro satélites não estavam segurados e valiam \$500 milhões. Testes de caminho [que exercitam o caminho de conversão] teriam descoberto o defeito, mas foram vetados por razões de orçamento."

Notícia de um jornal

18.4.3 Derivação de casos de teste

O método de teste de caminho base pode ser aplicado a um projeto procedimental ou ao código-fonte. Nesta seção, apresento o teste de caminho básico como uma série de passos. O procedimento *média* (*average*), mostrado em PDL na Figura 18.4, será usado como exemplo para ilustrar cada passo no método de projeto de casos de teste. Note que *média*, embora sendo um algoritmo extremamente simples, contém condições compostas e ciclos. Os seguintes passos podem ser aplicados para derivar o conjunto base:

- 1. Usando o projeto ou o código como base, desenhe o grafo de fluxo correspondente.** Um grafo de fluxo é criado usando os símbolos e regras de construção apresentados na Seção 18.4.1. De acordo com o PDL para *média* na Figura 18.4, é criado um grafo de fluxo enumerando-se os comandos PDL que serão mapeados por nós correspondentes do grafo de fluxo. O grafo de fluxo correspondente é mostrado na Figura 18.5.
- 2. Determine a complexidade ciclomática do diagrama de fluxo resultante. A complexidade ciclomática $V(G)$ é determinada aplicando-se os algoritmos descritos na Seção 18.4.2.** Deve-se notar que $V(G)$ pode ser determinado sem desenvolver um grafo de fluxo contando todos os comandos condicionais no PDL (para o procedimento *média*, o total de condições compostas é igual a dois) e somando 1. De acordo com a Figura 18.5,

$$V(G) = 6 \text{ regiões}$$

$$V(G) = 17 \text{ arestas} - 13 \text{ nós} + 2 = 6$$

$$V(G) = 5 \text{ nós predicaos} + 1 = 6$$

- 3. Determine um conjunto base de caminhos linearmente independentes.** O valor de $V(G)$ fornece o limite superior no número de caminhos linearmente independentes através da estrutura de controle do programa. No caso do procedimento *média*, esperamos especificar seis caminhos:

Caminho 1: 1-2-10-11-13

Caminho 2: 1-2-10-12-13

Caminho 3: 1-2-3-10-11-13

FIGURA 18.4

PDL com nós identificados

PROCEDURE average;

* Este procedimento calcula a média de 100 ou menos números situados entre valores limites; também calcula a soma e o total de números válidos.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;

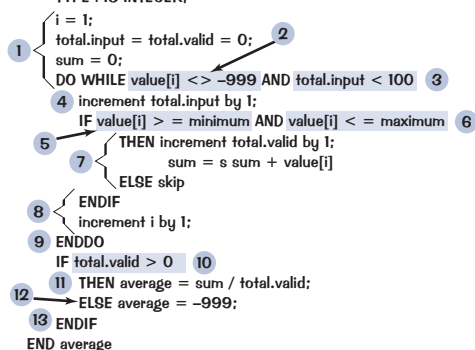
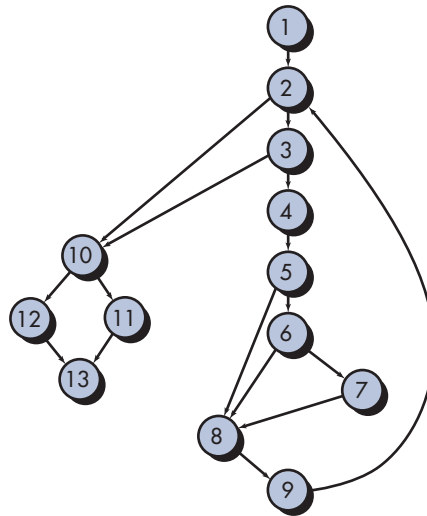


FIGURA 18.5

Grafo de fluxo para o procedimento médio



Caminho 4: 1-2-3-4-5-8-9-2-...

Caminho 5: 1-2-3-4-5-6-8-9-2-...

Caminho 6: 1-2-3-4-5-6-7-8-9-2-...

A reticência (...) após os caminhos 4, 5 e 6 indica que qualquer caminho através do resto da estrutura de controle é aceitável. Muitas vezes compensa identificar nós predicados como um auxílio na dedução de casos de teste. Nesse caso, os nós 2, 3, 5, 6 e 10 são nós predicados.

4. **Prepare casos de teste que vão forçar a execução de cada caminho do conjunto base.** Os dados devem ser escolhidos de forma que as condições nos nós predicados sejam definidas de forma apropriada à medida que cada caminho é testado. Cada caso de teste é executado e comparado com os resultados esperados. Depois que todos os casos de teste tiverem sido completados, o testador pode ter a certeza de que todos os comandos do programa foram executados pelo menos uma vez.

É importante notar que alguns caminhos independentes (por exemplo, caminho 1 em nosso exemplo) não podem ser testados de forma individual. Isso significa que a combinação de dados necessária para percorrer o caminho não pode ser conseguida no fluxo normal do programa. Nesses casos, esses caminhos são testados como parte de outro teste de caminho.

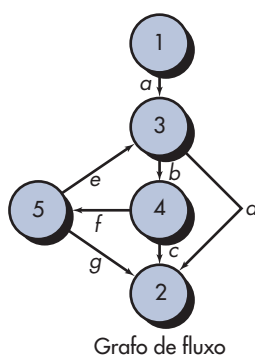
18.4.4 Matrizes de grafos

O procedimento para derivar o grafo de fluxo e até determinar um conjunto de caminhos base é passível de mecanização. Uma estrutura de dados, chamada de *matriz de grafos*, pode ser muito útil para o desenvolvimento de uma ferramenta de software que ajuda no teste do caminho base.

Uma matriz de grafo é uma matriz quadrada cujo tamanho (número de linhas e colunas) é igual ao número de nós no grafo de fluxo. Cada linha e coluna corresponde a um nó identificado, e as entradas da matriz correspondem a conexões (arestas) entre nós. Um exemplo simples de um grafo de fluxo e sua correspondente matriz de grafo [Bei90] é mostrado na Figura 18.6.

Observando a figura, cada nó do grafo de fluxo é identificado por números, enquanto cada aresta é identificada por letras. Uma letra na matriz corresponde à conexão entre dois nós. Por exemplo, o nó 3 está conectado ao nó 4 pela aresta *b*.

Até aqui, a matriz de grafo é nada mais do que uma representação tabular de um grafo de fluxo. No entanto, acrescentando um *peso de ligação* a cada entrada da matriz, a matriz de grafo pode se tornar uma poderosa ferramenta para avaliar a estrutura de controle de programa

FIGURA 18.6**Matriz de grafo**

Conectado ao nó		1	2	3	4	5
Nó	1			a		
2						
3			d		b	
4			c			f
5			g	e		

Matriz de grafo

durante o teste. O peso da ligação fornece informações adicionais sobre o fluxo de controle. Em sua forma mais simples, o peso da ligação é 1 (existe uma conexão) ou 0 (não existe uma conexão). Mas pesos de ligação podem ser atribuídos de acordo com outras propriedades mais interessantes:

- A probabilidade de que uma ligação (aresta) será executada.
- O tempo de processamento gasto para percorrer uma ligação
- A quantidade de memória necessária para percorrer uma ligação
- Os recursos necessários para percorrer uma ligação.

Beizer [Bei90] apresenta um tratamento completo de outros algoritmos matemáticos que podem ser aplicados às matrizes de grafos. Usando essas técnicas, a análise necessária para projeto de casos de teste pode ser parcial ou totalmente automatizada.

18.5 TESTE DE ESTRUTURA DE CONTROLE

“Dar mais atenção à execução dos testes do que ao seu projeto é um erro clássico.”

Brian Marick

A técnica de teste de caminho base descrita na Seção 18.4 é uma dentre várias técnicas para teste de estrutura de controle. Embora o teste de caminho base seja simples e altamente eficaz, ele sozinho não é suficiente. Nesta seção discutimos outras variações do teste de estrutura de controle. Elas ampliam a abrangência do teste e melhoram a qualidade do teste caixa-branca.

18.5.1 Teste de condição

Teste de condição [Tai89] é um método de projeto de caso de teste que exercita as condições lógicas contidas em um módulo de programa. Uma condição simples é uma variável booleana ou uma expressão relacional, possivelmente precedida por um operador NOT (¬). Uma expressão relacional toma a forma

$$E_1 <\text{operador relacional}> E_2$$

onde E_1 e E_2 são expressões aritméticas e $<\text{operador relacional}>$ é um dos seguintes operadores: $<$, $\leq$, $=$, $\neq$ (não igual), $>$, ou $\geq$. Uma *condição composta* é formada por duas ou mais condições simples, operadores booleanos e parênteses. Assumimos que os operadores booleanos permitidos em uma condição composta incluem OR ($\vee$), AND ($\&$) e NOT ($\neg$). Uma condição sem expressões relacionais é conhecida como expressão booleana.

Se uma condição é incorreta, então pelo menos um componente da condição é incorreto. Portanto, os tipos de erros em uma condição incluem erros de operador booleano (operadores booleanos incorretos, faltando ou extra), erros de variável booleana, erros de parênteses booleanos, erros de operador relacional e erros de expressão aritmética. O método de teste de condição focaliza o teste de cada condição no programa para garantir que ele não contenha erros.

PONTO-CHAVE

Os erros são muito mais comuns nas proximidades de condições lógicas do que próximos dos comandos de processamento sequencial.

"Bons testadores são mestres na arte de encontrar 'alguma coisa esquisita' e agir sobre ela."

Brian Marick

18.5.2 Teste de fluxo de dados

O método de teste de fluxo de dados [Fra93] seleciona caminhos de teste de um programa de acordo com as localizações de definições e usos de variáveis no programa. Para ilustrar a abordagem de teste de fluxo de dados, suponha que a cada comando em um programa é atribuído um número de comando único e que cada função não modifique seus parâmetros ou variáveis globais. Para um comando com S como seu número de comando,

$$\text{DEF}(S) = \{X \mid \text{comando } S \text{ contém uma definição de } X\}$$

$$\text{USE}(S) = \{X \mid \text{comando } S \text{ contém um uso de } X\}$$

Se o comando S for um *comando if* ou *de ciclo*, seu conjunto DEF está vazio e seu conjunto USE é baseado na condição do comando S . Dizemos que a definição da variável X no comando S é *considerada viva* no comando S' se existe um caminho do comando S para o comando S' que não contenha outra definição de X .

Uma *cadeia definição-uso (DU)* da variável X é da forma $[X, S, S']$, onde S e S' são números de instrução, X está em $\text{DEF}(S)$ e $\text{USE}(S')$, e a definição de X na instrução S está viva no comando S' .

Uma estratégia simples de teste de fluxo de dados é requerer que toda cadeia DU seja coberta pelo menos uma vez. Chamamos essa estratégia de *estratégia de teste DU*. Tem sido demonstrado que o teste DU não garante a abrangência de todos os desvios de um programa. No entanto, não é possível garantir que um desvio seja coberto pelo teste DU somente em raras situações como, por exemplo, construções *se-então-senão* nas quais a *parte então* não tem definição de qualquer variável e a *parte senão* não existe. Nessa situação, o desvio *senão* do comando *se não* é necessariamente coberto pelo teste DU.

18.5.3 Teste de ciclo

Os ciclos (loop) são a pedra fundamental da grande maioria dos algoritmos implementados em software. E ainda, muitas vezes prestamos pouca atenção enquanto executamos testes de software.

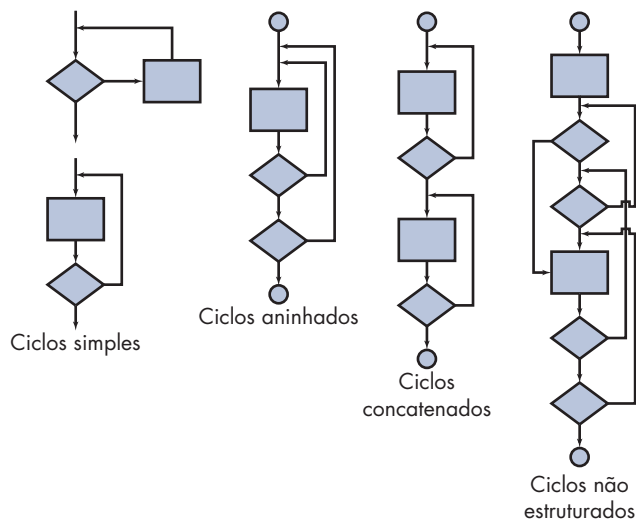
O *teste de ciclo* é uma técnica de teste caixa-branca que focaliza exclusivamente a validade das construções de ciclo. Podem ser definidas quatro diferentes classes de ciclos [Bei90]: ciclos simples, ciclos concatenados, ciclos aninhados e ciclos não estruturados (Figura 18.7).

Ciclos simples. O seguinte conjunto de testes pode ser aplicado a ciclos simples, onde n é o número máximo de passadas permitidas através do ciclo.



É irreal supor que o teste de fluxo de dados será usado extensivamente ao se testar um grande sistema. No entanto, ele pode ser usado especificamente em áreas do software que sejam suspeitas.

FIGURA 18.7
Classes de ciclos



1. Pular o ciclo inteiramente.
2. Somente uma passagem pelo ciclo.
3. Duas passagens pelo ciclo.
4. m passagens através do ciclo onde $m < n$.
5. $n - 1$, n , $n + 1$ passagens através do ciclo.

Ciclos aninhados. Se fôssemos estender a abordagem de teste de ciclos simples para ciclos aninhados, o número de testes possíveis cresceria geometricamente à medida que o nível de aninhamento aumentasse. O resultado seria um número impossível de testes. Beizer [Bei90] sugere uma abordagem que ajudará a reduzir o número de testes:

1. Comece pelo ciclo mais interno. Coloque todos os outros ciclos nos seus valores mínimos.
2. Faça os testes de ciclo simples para o ciclo mais interno mantendo, ao mesmo tempo, os ciclos externos em seus parâmetros mínimos de iteração (por exemplo, contador do ciclo). Acrescente outros testes para valores fora do intervalo ou excluídos.
3. Trabalhe para fora, fazendo testes para o próximo ciclo, mas mantendo todos os outros ciclos externos nos seus valores mínimos e outros ciclos aninhados com valores “típicos”.
4. Continue até que todos os ciclos tenham sido testados.



Você não pode testar ciclos não estruturados eficientemente. Reprojete-os.

Ciclos concatenados. Ciclos concatenados podem ser testados usando a abordagem definida para ciclos simples, se cada um for independente do outro. No entanto, se dois ciclos forem concatenados e a contagem para o ciclo 1 for usada como valor individual para o ciclo 2, então os ciclos não são independentes. Quando os ciclos não são independentes, é recomendada a abordagem aplicada a ciclos aninhados.

Ciclos não estruturados. Sempre que possível, essa classe de ciclos deverá ser redesenhada para refletir o uso das construções de programação estruturada (Capítulo 10).

18.6 TESTE CAIXA-PRETA

Teste caixa-preta, também chamado de *teste comportamental*, focaliza os requisitos funcionais do software. As técnicas de teste caixa-preta permitem derivar séries de condições de entrada que utilizarão completamente todos os requisitos funcionais para um programa. O teste caixa-preta não é uma alternativa às técnicas caixa-branca. Em vez disso, é uma abordagem complementar, com possibilidade de descobrir uma classe de erros diferente daquela obtida com métodos caixa-branca.

O teste caixa-preta tenta encontrar erros nas seguintes categorias: (1) funções incorretas ou faltando, (2) erros de interface, (3) erros em estruturas de dados ou acesso a bases de dados externas, (4) erros de comportamento ou de desempenho, e (5) erros de inicialização e término.

Diferentemente do teste caixa-branca, que é executado antecipadamente no processo de teste, o teste caixa-preta tende a ser aplicado durante estágios posteriores do teste (veja o Capítulo 17). Devido ao teste caixa-preta propositadamente desconsiderar a estrutura de controle, a atenção é focalizada no domínio das informações. Os testes são feitos para responder às seguintes questões:

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema é testado?
- Que classes de entrada farão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como as fronteiras de uma classe de dados é isolada?
- Que taxas e volumes de dados o sistema pode tolerar?
- Que efeito combinações específicas de dados terão sobre a operação do sistema?

“Errar é humano; encontrar um defeito é divino.”

Robert Dunn

? Que perguntas são respondidas pelos testes caixa-preta?

PONTO-CHAVE

Um grafo representa a relação entre objetos dados e objetos programa, permitindo criar casos de teste que procuram por erros associados com essas relações.

Aplicando técnicas caixa-preta, você cria uma série de casos de teste que satisfaz aos seguintes critérios [Mye79]: (1) casos de teste que reduzem, de um valor maior do que um, o número de casos de teste adicionais que devem ser projetados para atingir um teste razoável; e (2) casos de teste que lhe dizem alguma coisa sobre a presença ou ausência de classes de erros, em vez de um erro associado somente com o teste específico que se está fazendo.

18.6.1 Métodos de teste com base em grafo

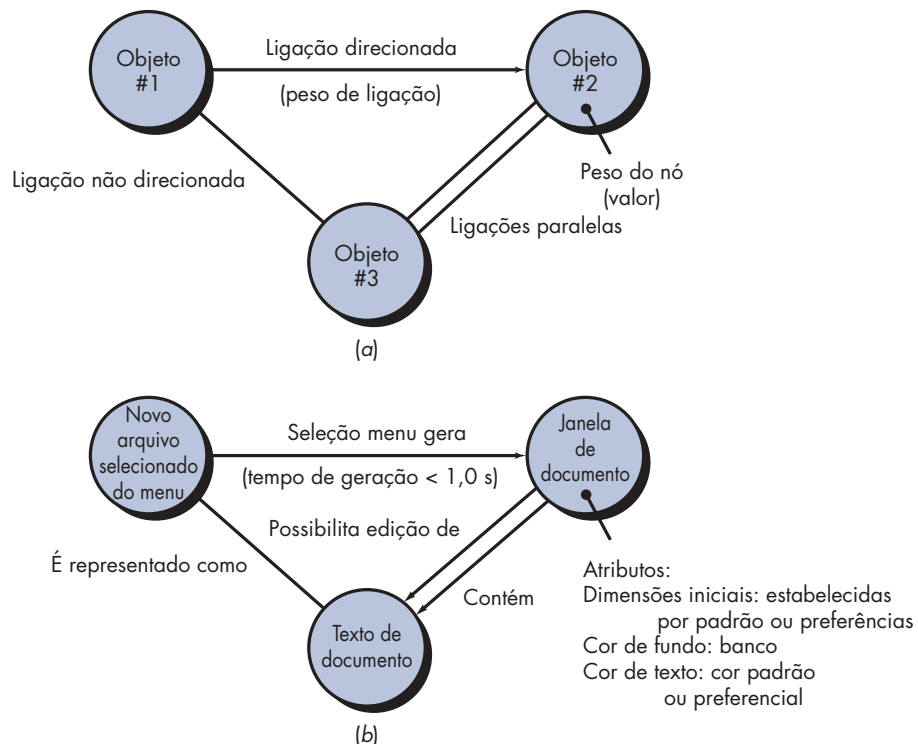
O primeiro passo no teste caixa-preta é entender os objetos⁵ que são modelados no software e as relações que unem esses objetos. Uma vez conseguido isso, o próximo passo é definir uma série de testes que verificam que “todos os objetos têm a relação esperada uns com os outros” [Bei95]. Colocado de outra forma, o teste do software começa criando um grafo de objetos importantes e suas relações e então imaginando uma série de testes que abrangerá o grafo de forma que cada objeto e relação sejam exercitados e os erros sejam descobertos.

Para executar esses passos, você começa criando um *grafo* — uma coleção de *nós* que representam objetos, *ligações* que representam as relações entre objetos, *pesos de nó* que descrevem as propriedades de um nó (por exemplo, o valor específico de um dado ou comportamento de estado), e pesos de ligação (*link weights*) que descrevem alguma característica de uma ligação.

A representação simbólica de um grafo é mostrada na Figura 18.8a. Os nós são representados como círculos unidos por ligações que assumem várias formas diferentes. Uma *ligação direta* (representada por uma seta) indica que uma relação se move apenas em uma direção. Uma *ligação bidirecional*, também chamada de *ligação simétrica*, significa que a relação se aplica em ambas as direções. *Ligações paralelas* são usadas quando várias relações diferentes são estabelecidas entre os nós do grafo.

FIGURA 18.8

(a) Notação de grafo; (b) exemplo simples



⁵ Nesse contexto, considera-se o termo *objeto* no sentido mais amplo possível. Abrange os objetos de dados, componentes tradicionais (módulos) e elementos de software orientados a objeto

Como um exemplo simples, considere uma parte de um grafo para uma aplicação de processador de texto (Figura 18.8b) na qual

Objeto 1 = **arquivoNovo** (seleção de menu)

Objeto 2 = **janelaDeDocumento**

Objeto 3 = **textoDeDocumento**

De acordo com a figura, uma seleção de menu em **arquivoNovo** gera uma janela de documento. O peso do nó de **janelaDeDocumento** fornece uma lista dos atributos de janela que devem ser esperados quando a janela é gerada. O *peso da ligação* indica que a janela deve ser gerada em menos de 1,0 segundo. Uma *ligação indireta* estabelece uma relação simétrica entre a seleção de menu **arquivoNovo** e **textoDeDocumento**, e *ligações paralelas* indicam relações entre **janelaDeDocumento** e **textoDeDocumento**. Na realidade, teria de ser gerado um grafo muito mais detalhado como um precursor para o projeto do caso de teste. Você pode então criar casos de teste atravessando o grafo e percorrendo cada uma das relações mostradas. Esses casos de teste são projetados numa tentativa de encontrar erros em qualquer uma das relações. Beizer [Bei95] descreve uma série de métodos de teste comportamental que utilizam grafos:

Modelagem de fluxo de transação. Os nós representam passos em alguma transação (por exemplo, os passos necessários para fazer uma reserva em uma empresa aérea usando um serviço on-line), e as ligações representam a conexão lógica entre os passos (por exemplo, **flightInformationInput** [entrada de informação sobre o voo] é seguido por *validation-AvailabilityProcessing* [processamento de validação da disponibilidade]). O grafo de fluxo de dados (Capítulo 7) pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de estado finito. Os nós representam diferentes estados observáveis pelo usuário do software (por exemplo, cada uma das “telas” que aparecem quando um atendente recebe um pedido por telefone), e as ligações representam as transições que ocorrem para mover de um estado para outro (por exemplo, **orderInformation** [informação sobre o pedido] é verificada durante *inventoryAvailabilityLook-up* [consulta da disponibilidade no inventário] e é seguida pela entrada **customerBillingInformation** [informações para faturamento]). O grafo de estado (Capítulo 7) pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de fluxo de dados. Os nós são objetos dados e as ligações são as transformações que ocorrem para traduzir um objeto dados em outro. Por exemplo, o nó imposto retido (FICA tax withheld – **FTW**) é calculado com base no salário bruto (Gross wages – **GW**) usando a relação, **FTW** = 0,62 x **GW**.

Modelagem no tempo. Os nós são objetos de programa, e as ligações são conexões sequenciais entre aqueles objetos. *Pesos de ligação* são usados para especificar os tempos de execução necessários enquanto o programa é executado.

Uma discussão detalhada de cada um desses métodos de teste com base em diagrama vai além do escopo deste livro. Se você estiver interessado, veja [Bei95] para uma descrição detalhada.



As classes de entrada são conhecidas relativamente no início da gestão de qualidade. Por essa razão, comece pensando sobre o particionamento de equivalência logo que o projeto for criado.

18.6.2 Particionamento de equivalência

Particionamento de equivalência é um método de teste caixa-preta que divide o domínio de entrada de um programa em classes de dados a partir das quais podem ser criados casos de teste. Um caso de teste ideal descobre sozinho uma classe de erros (por exemplo, processamento incorreto de todos os dados de caracteres) que poderia, de outro modo, requerer que fossem executados muitos casos de teste até que o erro geral aparecesse.

O projeto de casos de teste para particionamento de equivalência tem como base a avaliação das *classes de equivalência* para uma condição de entrada. Usando conceitos introduzidos na seção anterior, se um conjunto de objetos pode ser vinculado por relações simétricas, transitivas e reflexivas, uma classe de equivalência estará presente [Bei95]. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente,

uma condição de entrada é um valor numérico específico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Classes de equivalência podem ser definidas de acordo com as seguintes regras:

1. Se uma condição de entrada especifica um intervalo, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
2. Se uma condição de entrada requer um valor específico, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
3. Se uma condição de entrada especifica um membro de um conjunto, são definidas uma classe de equivalência válida e uma classe de equivalência inválida.
4. Se uma condição de entrada for booleana, são definidas uma classe válida e uma inválida.

Aplicando as diretrizes para a derivação de classes de equivalência, podem ser desenvolvidos e executados casos de teste para o domínio de entrada de cada item de dado. Os casos de teste são selecionados de maneira que o máximo de atributos de uma classe de equivalência sejam exercitados ao mesmo tempo.

18.6.3 Análise de valor limite

Um número maior de erros ocorre nas fronteiras do domínio de entrada e não no “centro”. É por essa razão que foi desenvolvida a *análise do valor limite* (*boundary value analysis* - BVA) como uma técnica de teste. A análise de valor limite leva a uma seleção de casos de teste que utilizam valores limites.

A análise de valor limite é uma técnica de projeto de casos de teste que complementa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA conduz à seleção de casos de teste nas “bordas” da classe. Em vez de focalizar somente condições de entrada, a BVA obtém casos de teste também a partir do domínio de saída [Mye79].

As diretrizes para a BVA são similares, em muitos aspectos, àquelas proporcionadas para o particionamento de equivalência:

1. Se uma condição de entrada especifica um intervalo limitado por valores a e b , deverão ser projetados casos de teste com valores a e b imediatamente acima e abaixo de a e b .
2. Se uma condição de entrada especifica um conjunto de valores, deverão ser desenvolvidos casos de teste que usam os números mínimo e máximo. São testados também valores imediatamente acima e abaixo dos valores mínimo e máximo.
3. Aplique as diretrizes 1 e 2 às condições de saída. Por exemplo, suponha que um programa de análise de engenharia precisa ter como saída uma tabela de temperatura *versus* pressão. Deverão ser projetados casos de teste para criar um relatório de saída que produza o número máximo (e mínimo) permitido de entradas da tabela.
4. Se as estruturas de dados internas do programa prescreveram fronteiras (por exemplo, uma tabela tem um limite definido de 100 entradas), não esqueça de criar um caso de teste para exercitar a estrutura de dados na fronteira.

Até certo ponto, a maioria dos engenheiros de software executa intuitivamente a BVA. Aplicando essas diretrizes, o teste de fronteira será mais completo, tendo assim uma possibilidade maior de detecção de erro.

18.6.4 Teste de matriz ortogonal

Há muitas aplicações nas quais o domínio de entrada é relativamente limitado. Isso significa que o número de parâmetros de entrada é pequeno e os valores que cada um dos parâmetros pode tomar estão claramente limitados. Quando esses números são muito pequenos (por exem-

? Como definir classes de equivalência para teste?

“Uma maneira eficaz de testar código é exercitá-lo em suas fronteiras naturais.”

Brian Kernighan

PONTO-CHAVE

A BVA estende o particionamento de equivalência focalizando os dados nas “fronteiras” de uma classe de equivalência.

plo, três parâmetros de entrada assumindo três valores discretos cada um), é possível considerar cada permutação de entrada e testar exaustivamente o domínio de entrada. No entanto, à medida que cresce o número de valores de entrada e o número de valores discretos para cada item de dado aumenta, o teste exaustivo torna-se impraticável ou impossível.

O teste de matriz ortogonal pode ser aplicado a problemas nos quais o domínio de entrada é relativamente pequeno, mas muito grande para acomodar o teste exaustivo. O método de teste de matriz ortogonal é particularmente útil para encontrar erros associados com falhas de regiões — uma categoria de erro associada com lógica defeituosa em um componente de software.

Para ilustrar a diferença entre teste de matriz ortogonal e abordagens mais convencionais do tipo “uma entrada de cada vez”, considere um sistema que tenha três itens de entrada, X, Y e Z. Cada um desses itens de entrada tem três valores discretos associados com ele. Existem $3^3 = 27$ casos de teste possíveis. Phadke [Pha97] sugere uma visualização geométrica dos casos de teste possíveis associados com X, Y e Z ilustrada na Figura 18.9. Olhando a figura, um item de entrada de cada vez pode ser variado em sequência ao longo de cada eixo de entrada. Isso resulta em uma abrangência relativamente limitada do domínio de entrada (representado pelo cubo da esquerda na figura).

Quando ocorre o teste de matriz ortogonal, é criada uma matriz ortogonal L9 de casos de teste. O conjunto ortogonal L9 tem uma “propriedade de balanceamento” [Pha97]. Ou seja, casos de teste (representados por pontos escuros na figura) são “dispersos uniformemente através do domínio do teste”, conforme ilustra o cubo da direita na Figura 18.9. A cobertura de teste ao longo do domínio de entrada é mais completa.

Para ilustrar o uso da matriz ortogonal L9, considere a função *envie* para uma aplicação de fax. São passados quatro parâmetros, P1, P2, P3, e P4, para a função *send*. Cada parâmetro assume três valores discretos. Por exemplo, P1 assume os valores:

- P1 = 1, enviar agora
- P1 = 2, enviar após 1 hora
- P1 = 3, enviar depois da meia-noite

P2, P3, e P4 também assumiriam valores 1, 2 e 3, significando outras funções de envio.

Se fosse escolhida a estratégia de teste “um item de entrada de cada vez”, seria especificada a seguinte sequência (P1, P2, P3, P4) de testes: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), e (1, 1, 1, 3). Phadke [Pha97] avalia esses casos de teste afirmando:

Esses casos de teste somente são úteis quando se tem certeza de que esses parâmetros de teste não interagem. Eles podem detectar falhas de lógica onde um único valor de parâmetro faz o software falhar. Essas falhas são chamadas de *falhas de modo simples* (*single mode faults*). Esse método não pode detectar falhas lógicas que causam mau funcionamento quando dois ou mais parâmetros assumem simultaneamente certos valores; isto é, ele não pode detectar quaisquer interações. Portanto, sua habilidade em detectar falhas é limitada.

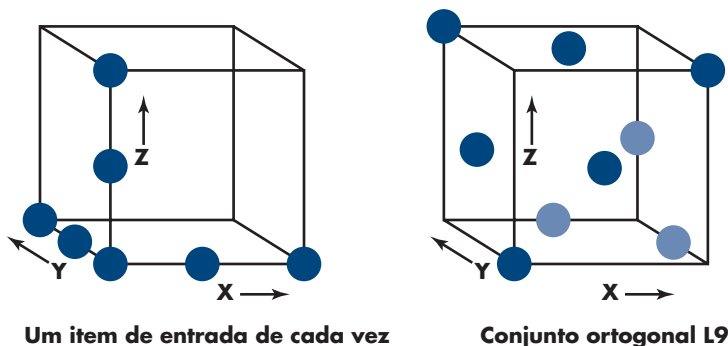
PONTO-CHAVE

A BVA estende o particionamento de equivalência focalizando os dados nas “fronteiras” de uma classe de equivalência.

FIGURA 18.9

Uma visualização geométrica dos casos de teste

Fonte: (Pha97)



Dado o número relativamente pequeno de parâmetros de entrada e valores discretos, o teste exaustivo é possível. O número de testes necessários é $3^4 = 81$, grande, porém controlável. Todas as falhas associadas com permutação de itens de dados seriam encontradas, mas o trabalho necessário é relativamente alto.

A abordagem de teste de matriz ortogonal permite obter boa abrangência de teste com bem menos casos de teste do que a estratégia exaustiva. Uma matriz ortogonal L9 para a função *envie* da aplicação de fax está ilustrada na Figura 18.10.

FIGURA 18.10

Uma matriz ortogonal L9

Casos de teste	Parâmetros de teste			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

FERRAMENTAS DO SOFTWARE



Projeto de caso de teste

Objetivo: auxiliar a equipe de software no desenvolvimento de uma série completa de casos de teste, tanto para teste caixa-preta quanto para caixa-branca.

Mecanismos: essas ferramentas se classificam em duas categorias principais: ferramentas de teste estático e ferramentas de teste dinâmico. São usadas na indústria três tipos diferentes de ferramentas de teste estático: ferramentas de teste baseadas em código, linguagens especializadas de teste e ferramentas de teste baseadas em requisitos. As ferramentas de teste baseadas em código aceitam código-fonte como entrada e executam uma série de análises que resultam na geração de casos de teste. As linguagens de teste especializadas (por exemplo, ATLAS) permitem a um engenheiro de software escrever especificações detalhadas de teste que descrevem cada caso de teste e as lógicas para sua execução. Ferramentas de teste baseadas em requisitos isolam requisitos específicos de usuário e sugerem casos de teste (ou classes de testes) que exercitarão os requisitos. As ferramentas de teste dinâmico interagem com um programa em execução, verificando amplitude do caminho, testando asserções sobre o valor de variáveis específicas, e instrumentando o fluxo de execução do programa de qualquer modo.

Ferramentas representativas:⁶

McCabeTest, desenvolvida pela McCabe & Associates (www.mccabe.com), implementa uma variedade de técnicas de teste de caminho derivadas de uma avaliação da complexidade ciclomática e de outras métricas de software.

TestWorks, desenvolvida pela Software Research, Inc. (www.soft.com/Products), é uma série completa de ferramentas de teste automáticas que ajudam no projeto de casos de teste para software desenvolvido em C/C++ e Java e proporciona suporte para teste de regressão.

T-VEC Test Generation System, desenvolvida pela T-VEC Technologies (www.t-vec.com), é uma ferramenta que suporta teste de unidade, integração e validação ajudando no projeto de casos de teste usando informações contidas em uma especificação de requisitos Orientados a Objeto (OO).

e-TEST Suite, desenvolvida pela Empirix, Inc. (www.empirix.com), abrange uma série completa de ferramentas para testar WebApps, incluindo ferramentas que ajudam no projeto de casos de teste e planejamento de teste.

⁶ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Phadke [Pha97] avalia o resultado dos testes usando a matriz ortogonal L9 da seguinte maneira:

Detecte e isole todas as falhas de modo simples. Uma falha de modo simples é um problema consistente com qualquer nível de qualquer parâmetro isolado. Por exemplo, se todos os casos de teste de fator $P1 = 1$ causam uma condição de erro, trata-se de uma falha de modo simples. Nesse exemplo, os testes 1, 2 e 3 [Figura 18.10] mostrarão erros. Analisando as informações sobre quais testes mostram erros, pode-se identificar que valores de parâmetros causam a falha. Nesse exemplo, observando que os testes 1, 2 e 3 causam um erro, podemos isolar [o processamento lógico associado com “enviar agora” ($P1 = 1$)] como origem do erro. Esse isolamento da falha é importante para poder corrigi-la.

Detecte todas as falhas de modo duplo. Se existe um problema consistente quando níveis específicos de dois parâmetros ocorrem juntos, chamamos isso de *falha de modo duplo* (*double mode fault*). Sem dúvida, uma falha de modo duplo é uma indicação de incompatibilidade do par ou interações danosas entre dois parâmetros de teste.

Falhas multimodo. Matrizes ortogonais [do tipo mostrado] somente podem garantir a detecção de falhas de modo simples e duplo. No entanto, muitas falhas multimodo também são detectadas por esses testes.

Uma discussão detalhada do teste de matriz ortogonal pode ser encontrada em [Pha89].

18.7 TESTE BASEADO EM MODELOS

“É bastante difícil encontrar um erro no seu código quando você o está procurando; é ainda mais difícil quando você assume que o seu software é isento de erros.”

Steve McConnell

Teste baseado em modelo (*Model-based testing* - MBT) é uma técnica de teste caixa-preta que usa informações contidas no modelo de requisitos como base para a geração de casos de teste. Em muitos casos, as técnicas baseadas em modelo usam diagramas de estado UML, um elemento do modelo comportamental (Capítulo 7), como base para o projeto de casos de teste.⁷ A técnica MBT requer cinco passos:

- 1. Analise um modelo comportamental existente para o software ou crie um.** Lembre-se de que o *modelo comportamental* indica como o software responderá a eventos ou estímulos externos. Para criar o modelo, você deverá executar os passos discutidos no Capítulo 7: (1) avaliar todos os casos de uso para entender completamente a sequência de interação dentro do sistema, (2) identificar eventos que controlam a sequência de interação e entender como esses eventos se relacionam com objetos específicos, (3) criar uma sequência para cada caso de uso, (4) criar um diagrama de estado UML para o sistema (por exemplo, veja a Figura 7.6), e (5) rever o modelo comportamental para verificar exatidão e consistência.
- 2. Percorra o modelo comportamental e especifique as entradas que forçarão o software a fazer a transição de um estado para outro.** As entradas irão disparar eventos que farão a transição ocorrer.
- 3. Reveja o modelo comportamental e observe as saídas esperadas à medida que o software faz a transição de um estado para outro.** Lembre-se de que cada transição de estado é disparada por um evento e que, em consequência da transição, alguma função é chamada e saídas são criadas. Para cada conjunto de entradas (casos de teste) que você especificou no passo 2, especifique as saídas esperadas como elas são caracterizadas no modelo comportamental. “Uma hipótese fundamental desse teste é que há algum mecanismo, uma *previsão de teste* (*test oracle*), que determinará se os resultados da execução de um teste são ou não corretos” [DAC03]. Essencialmente, uma previsão de teste estabelece a base para qualquer determinação da precisão da saída. Em muitos casos, a previsão é o modelo de requisitos, mas poderia ser também outro documento ou aplicação, dados gravados em algum lugar, ou mesmo um especialista humano.

⁷ O teste baseado em modelo também pode ser usado quando os requisitos de software são representados por tabelas de decisões, gramáticas, ou cadeias de Markov [DAC03].

4. **Execute os casos de teste.** Os testes podem ser executados manualmente ou pode ser criado um *script* de teste e executado usando uma ferramenta de teste.
5. **Compare os resultados real e esperado e tome a ação corretiva necessária.**

O MBT ajuda a descobrir erros no comportamento do software, e consequentemente, é extremamente útil ao testar aplicações acionadas por eventos.

18.8 TESTE PARA AMBIENTES, ARQUITETURAS E APLICAÇÕES ESPECIALIZADOS

Às vezes, é possível assegurar diretrizes e abordagens únicas quando são considerados ambientes, arquiteturas e aplicações especializados. Embora as técnicas de teste discutidas anteriormente neste capítulo e nos Capítulos 19 e 20 possam muitas vezes ser adaptadas a situações específicas, é importante considerar individualmente suas necessidades especiais.

18.8.1 Testando GUIs

As Interfaces Gráficas de Usuário (*Graphical user interfaces* - GUIs) apresentam desafios interessantes de teste. Devido aos componentes reutilizáveis serem agora parte comum do desenvolvimento de ambientes de GUI, a criação de interfaces de usuário tornou-se mais rápida e mais precisa (Capítulo 11). Mas, ao mesmo tempo, a complexidade das GUIs tem crescido, resultando em dificuldades maiores no projeto e execução de casos de teste.

Como muitas GUIs modernas têm a mesma aparência e comportamento, é possível criar uma série de testes padronizados. Grafos de modelagem de estado finito podem ser usados para criar uma série de testes que cuidam de objetos específicos de dados e programa relevantes para a GUI. Essa técnica de teste baseada em modelo foi discutida na Seção 18.7.

Devido ao grande número de permutações associadas com operações GUI, o teste de GUI deverá ser abordado usando ferramentas automatizadas. Uma ampla variedade de técnicas de teste GUI surgiu no mercado nos últimos anos.⁸

18.8.2 Teste de arquiteturas cliente-servidor

A natureza distribuída de ambientes cliente-servidor, os aspectos de desempenho associados com processamento de transações, a presença potencial de muitas plataformas de hardware diferentes, a complexidade das comunicações em rede, a necessidade de servir a múltiplos clientes a partir de uma base de dados centralizada (ou em alguns casos, distribuída), e os requisitos de coordenação impostos sobre o servidor, tudo se combina para tornar o teste de arquiteturas cliente-servidor e o software que reside nelas consideravelmente mais difícil do que para aplicações individualizadas. De fato, estudos recentes indicam um aumento significativo no tempo e custo de teste quando são desenvolvidos ambientes cliente-servidor.

Em geral, o teste de software cliente-servidor ocorre em três níveis diferentes: (1) Aplicações cliente individuais são testadas em um modo “desconectado”; a operação do servidor e a rede subjacente não são consideradas. (2) O software cliente e as aplicações servidor associadas são testados em harmonia, mas as operações de rede não são exercitadas explicitamente. (3) É testada a arquitetura completa cliente-servidor, incluindo operação e desempenho de rede.

Embora sejam executados muitos tipos diferentes de testes em cada um desses níveis de detalhe, as abordagens de teste a seguir são muitas vezes encontradas para aplicações cliente-servidor:

- **Testes de função de aplicação.** A funcionalidade das aplicações cliente é testada usando os métodos discutidos anteriormente neste capítulo e nos Capítulos 19 e 20. Essencialmente, a aplicação é testada no modo individual (*stand-alone*), na tentativa de descobrir erros em sua operação.

“O tópico de testes é uma área na qual existe uma boa dose de pontos comuns entre sistema tradicional e sistemas cliente/servidor.”

Kelley Bourne

WebRef

Informações e recursos úteis para teste cliente-servidor podem ser encontrados em www.csstechnologies.com.

? Que tipos de testes são executados para sistemas cliente-servidor?

⁸ Centenas, se não milhares de recursos de ferramentas de teste GUI podem ser avaliados na Internet. Um bom ponto de partida para ferramentas de código aberto (*open-source*) é www.opensourcetesting.org/functional.php.

- **Testes de servidor.** São testadas as funções de coordenação e gerenciamento de dados do servidor. É considerado também o desempenho do servidor (tempo de resposta geral e taxa de saída de dados).
- **Testes de base de dados.** É testada a exatidão e a integridade dos dados armazenados pelo servidor. São examinadas as transações postadas por aplicações cliente para assegurar que os dados estejam corretamente armazenados, atualizados e acessados. É testado também o arquivamento.
- **Testes de transação.** É criada uma série de testes para garantir que cada classe de transações seja processada de acordo com os requisitos. Os testes focalizam a exatidão do processamento e também os problemas de desempenho (por exemplo, tempos de processamento de transação e volume de transação).
- **Testes de comunicação de rede.** Esses testes verificam se as comunicações entre os nós da rede ocorrem corretamente e se a passagem de mensagens, transações e tráfego relacionado com a rede ocorrem sem erro. Podem ser executados também testes de segurança de rede como parte desses testes.

Para completar essas abordagens de teste, Musa [Mus93] recomenda o desenvolvimento de *perfis operacionais* derivados de cenários de uso cliente-servidor.⁹ Um perfil operacional indica como diferentes tipos de usuários interoperam com o sistema cliente-servidor. Isto é, os perfis fornecem um “padrão de uso” que pode ser aplicado quando os testes são projetados e executados. Por exemplo, para um tipo particular de usuário, que porcentagem das transações será de consultas? de atualizações? de pedidos?

Para desenvolver o perfil operacional, é necessário criar um conjunto de cenários que seja similar aos casos de uso (Capítulos 5 e 6). Cada cenário esclarece quem, onde, o que e por quê: quem é o usuário, onde (na arquitetura física cliente-servidor) ocorre a interação do sistema, qual é a transação e por que ela ocorreu. Cenários podem ser criados usando técnicas de extração de requisitos (Capítulo 5) ou por meio de discussões menos formais com os usuários finais. O resultado, no entanto, deverá ser o mesmo. Cada cenário deverá fornecer uma indicação das funções do sistema que serão necessárias para atender a um usuário em particular, a ordem na qual aquelas funções são necessárias, o tempo e resposta esperados e a frequência com que cada função é usada. Esses dados são então combinados (para todos os usuários) para criar o perfil operacional. Em geral, o trabalho de teste e o número de casos de teste a ser executado são alocados para cada cenário de usuário com base na frequência de uso e na criticidade das funções executadas.

18.8.3 Testando a documentação e os recursos de ajuda

O termo *teste de software* passa a imagem de grande quantidade de casos de teste preparados para exercitar programas de computador e os dados que manipulam. Recordando a definição de software apresentada no Capítulo 1, é importante notar que o teste também deve se estender ao terceiro elemento da configuração do software — a documentação.

Erros na documentação podem ser tão devastadores para a aceitação dos programas quanto os erros nos dados ou no código-fonte. Nada é mais frustrante do que seguir um guia do usuário ou um recurso de ajuda on-line exatamente e obter resultados ou comportamentos que não coincidem com aqueles previstos pela documentação. É por essa razão que o teste da documentação deve ser parte significativa de todo plano de teste de software.

O teste da documentação pode ser feito em duas fases. A primeira fase, a revisão técnica (Capítulo 15), examina o documento quanto à clareza de edição. A segunda fase, o teste ao vivo, usa a documentação em conjunto com o programa real.

⁹ Deve-se notar que os perfis operacionais podem ser usados no teste de todos os tipos de arquiteturas de sistema, não apenas na arquitetura cliente-servidor.

Surpreendentemente, um teste ao vivo para documentação pode ser feito usando técnicas análogas a muitos métodos de teste caixa-preta discutidos anteriormente. Teste baseado em diagrama pode ser usado para descrever o uso do programa; particionamento de equivalência e análise de valor limite podem ser usados para definir várias classes de entradas e interações associadas. MBT pode ser usado para garantir que o comportamento documentado e o comportamento real coincidam. O uso do programa é então acompanhado com base da utilização da documentação.



Teste de Documentação

As seguintes questões deverão ser respondidas durante o teste de documentação e/ou recurso de ajuda:

- A documentação descreve com precisão como proceder em cada modo de utilização?
- A descrição de cada sequência de interação é precisa?
- Os exemplos são precisos?
- A terminologia, as descrições dos menus e as respostas do sistema são consistentes com o programa real?
- É relativamente fácil localizar diretrizes dentro da documentação?
- A solução de problemas pode ser obtida facilmente com a documentação?
- O sumário e o índice do documento são bons, exatos e completos?
- O estilo do documento (layout, fontes, recuos, gráficos) conduz ao entendimento e rápida assimilação das informações?
- Todas as mensagens de erro do software que aparecem para o usuário estão descritas em mais detalhes no documento? As ações a ser tomadas em consequência de uma mensagem de erro estão claramente delineadas?
- Se forem usadas ligações de hipertexto, elas são precisas e completas?
- Se for usado hipertexto, o estilo de navegação é apropriado para as informações requeridas?

A única maneira viável de responder a essas questões é através de uma consultoria independente (por exemplo, usuários escolhidos) para testar a documentação no contexto do uso do programa. Todas as discrepâncias são anotadas, e as áreas do documento que apresentam ambiguidade ou deficiências são marcadas para ser reescritas.

INFORMAÇÕES

18.8.4 Teste para sistema em tempo real

A natureza assíncrona, dependente do tempo, de muitas aplicações em tempo real acrescenta um elemento novo e potencialmente difícil ao conjunto de testes — o tempo. O projetista do caso de teste tem de considerar não apenas os casos de teste convencionais mas também a manipulação de eventos (o processamento de interrupção), a temporização dos dados e o paralelismo das tarefas (processos) que manipulam os dados. Em muitas situações, dados de teste fornecidos quando o sistema em tempo real está em um estado resultarão em um processamento correto, enquanto os mesmos dados fornecidos quando o sistema está em um estado diferente podem resultar em erro.

Por exemplo, o software em tempo real que controla uma nova fotocopadora aceita interrupções do operador (o operador da máquina pressiona teclas de controle como RESET ou ESCURECER) sem resultar em erro quando a máquina está fazendo cópias (no estado “copiando”). Essas mesmas interrupções do operador, se forem acionadas quando a máquina está no estado “atolada” (*jammed*), causam a perda do código de diagnóstico que indicava a localização do enrosco (um erro).

Além disso, a relação íntima que existe entre um software em tempo real e seu ambiente de hardware pode também causar problemas de teste. Testes de software devem levar em consideração o impacto das falhas do hardware sobre o processamento do software. Essas falhas podem ser extremamente difíceis de simular realisticamente.

Métodos de projeto de casos de teste abrangentes para sistemas em tempo real continuam evoluindo. No entanto, pode-se propor uma estratégia geral de quatro passos:

- **Teste de tarefa.** O primeiro passo no teste de software de tempo real é testar cada tarefa independentemente. Testes convencionais são projetados para cada tarefa e executados

 Qual é uma estratégia eficaz para testar um sistema em tempo real?

independentemente durante esses testes. Testes de tarefa descobrem erros em lógica e funções, mas não em temporização ou comportamento.

- **Teste comportamental.** Usando modelos de sistema criados com ferramentas automáticas, é possível simular o comportamento de um sistema em tempo real e examinar seu comportamento em consequência de eventos externos. Essas atividades de análise podem servir como base para o projeto de casos de teste executados quando o software de tempo real é criado. Usando uma técnica que é similar ao particionamento de equivalência (Seção 18.6.2), eventos (por exemplo, interrupções, sinais de controle) são classificados para teste. Por exemplo, eventos para a fotocopadora podem ser interrupções de usuário (por exemplo, zerar o contador), interrupções mecânicas (por exemplo, papel enroscado), interrupções de sistema (por exemplo, pouco toner), e modos de falha (por exemplo, rolo superaquecido). Cada um desses eventos é testado individualmente, e o comportamento do sistema executável é examinado para detectar erros que ocorrem em consequência do processamento associado com esses eventos. O comportamento do modelo de sistema (desenvolvido durante a atividade de análise) e o software executável podem ser comparados quanto ao desempenho. Uma vez testada cada classe de eventos, os eventos são apresentados ao sistema em ordem aleatória e com frequência aleatória. O comportamento do software é examinado para detectar erros de comportamento.
- **Teste intertarefas.** Uma vez isolados os erros em tarefas individuais e no comportamento do sistema, o teste passa para os erros relacionados com o tempo. Tarefas assíncronas que devem comunicar-se umas com as outras são testadas com diferentes taxas de dados e carga de processamento para determinar se ocorrerão erros de sincronização intertarefas. Além disso, tarefas que se comunicam via fila de mensagens ou armazenamento de dados são testadas para descobrir erros no dimensionamento dessas áreas de armazenamento.
- **Teste de sistema.** O software e o hardware são integrados, e é feita uma gama completa de testes do sistema na tentativa de descobrir erros na interface software-hardware. A maioria dos sistemas em tempo real processa interrupções. Portanto, é essencial testar a manipulação desses eventos booleanos. Usando o diagrama de estado (Capítulo 7), o testador desenvolve uma lista de todas as interrupções possíveis e o processamento que ocorre em consequência das interrupções. São planejados, então, testes para avaliar as seguintes características do sistema:
 - São atribuídas prioridades corretas às interrupções e elas são manipuladas corretamente?
 - O processamento de cada interrupção é manipulado corretamente?
 - O desempenho (por exemplo, tempo de processamento) de cada procedimento de manipulação de interrupção está conforme os requisitos?
 - A chegada de um alto volume de interrupções em instantes críticos cria problemas em função ou desempenho?

Além disso, deverão ser testadas as áreas globais de dados usadas para transferir informações como parte do processamento de interrupção, para avaliar o potencial de geração de efeitos colaterais.

18.9 PADRÕES PARA TESTE DE SOFTWARE

WebRef

Um catálogo de padrões de teste de software pode ser encontrado em www.rbsc.com/pages/TestPatternList.htm.

O uso de padrões como um mecanismo para descrever soluções para problemas específicos de projeto foi discutido no Capítulo 12. Mas os padrões também podem ser usados para propor soluções para outras situações de engenharia de software — nesse caso, o teste de software. Os *padrões de teste* descrevem problemas comuns de teste e soluções que podem ajudar a lidar com eles.

Os padrões de teste fornecem não apenas diretrizes úteis no início das atividades de teste, mas também trazem três benefícios adicionais descritos por Marick [Mar02]:

PONTO-CHAVE

A BVA estende o particionamento de equivalência focalizando os dados nas “fronteiras” de uma classe de equivalência.

1. Eles [os padrões] fornecem um vocabulário para solucionadores de problemas. “Ei, sabe, deveríamos usar um Objeto nulo.”
2. Eles focalizam a atenção nas forças que estão por trás de um problema. Isso permite que os projetistas [de casos de teste] entendam melhor quando e por que uma solução se aplica.
3. Eles estimulam o pensamento iterativo. Cada solução cria um novo contexto no qual novos problemas podem ser resolvidos.

Embora esses benefícios sejam “leves”, não devem ser menosprezados. Grande parte do teste de software, inclusive durante a década passada, tem sido uma atividade *ad hoc*. Se os padrões de teste podem ajudar uma equipe de software a se comunicar sobre testes de forma mais eficaz, a entender as forças motivadoras que conduzem a uma abordagem específica para o teste, e a abordar o projeto de testes como uma atividade evolucionária na qual cada iteração resulta em um conjunto mais completo de casos de teste, então os padrões realmente dão uma grande contribuição.

Os padrões de teste são descritos de maneira muito semelhante aos padrões de projeto (Capítulo 12). Já foram propostos dezenas de padrões de testes na literatura (por exemplo, [Mar02]). Os três padrões a seguir (apresentados apenas de uma forma superficial) fornecem exemplos representativos:

WebRef

Padrões que descrevem organização de teste, eficiência, estratégia e solução de problemas podem ser encontrados em: www.testing.com/test-patterns/patterns/.

Nome do padrão: **Teste aos pares**

Resumo: Um padrão orientado a processo, o **teste aos pares** descreve uma técnica que é análoga à programação aos pares (Capítulo 3), na qual dois testadores trabalham em conjunto para projetar e executar uma série de testes que podem ser aplicados a atividades de teste de unidade, integração ou validação.

Nome do padrão: **Interface de teste separada**

Resumo: Há necessidade de testar todas as classes em um sistema orientado a objeto, incluindo “classes internas” (isto é, classes que não expõem qualquer interface fora do componente que as utilizou). O padrão **Interface de teste separada** descreve como criar “uma interface de teste que pode ser usada para descrever testes específicos em classes que são visíveis somente internamente a um componente” [Lan01].

Nome do padrão: **Teste de cenário**

Resumo: Uma vez feitos os testes de unidade e de integração, é necessário determinar se o software se comportará ou não de maneira que satisfaça aos usuários. O padrão **Teste de cenário** descreve uma técnica para exercitar o software a partir do ponto de vista do usuário. Uma falha nesse nível indica que o software deixou de atender aos requisitos visíveis para o usuário [Kan01].

Uma discussão clara dos padrões de teste está além do escopo deste livro. Se você estiver interessado em mais detalhes, veja em [Bin99] e [Mar02] informações adicionais sobre esse importante tópico.

18.10 RESUMO

O objetivo primário do projeto de caso de teste é criar uma série de testes que tenha a mais alta probabilidade de descobrir erros no software. Para conseguir esse objetivo, são usadas duas categorias diferentes de técnicas de projeto de caso de teste: teste caixa-branca e teste caixa-preta.

Os testes caixa-branca focalizam a estrutura de controle do programa. São criados casos de teste para assegurar que todas as instruções no programa foram executadas pelo menos uma vez durante o teste e que todas as condições lógicas foram exercitadas. O teste de caminho base, uma técnica caixa-branca, usa diagramas de programa (ou matrizes gráficas) para derivar o conjunto de testes linearmente independentes que garantirão abranger todas as instruções. O teste de condições e de fluxo de dados exercita mais a lógica do programa, e o teste de ciclos complementa outras técnicas caixa-branca fornecendo um procedimento para exercitar ciclos com vários graus de complexidade.

Hetzel [Het84] descreve o teste caixa-branca como “teste no pequeno”. Sua implicação é que os testes caixa-branca que foram considerados neste capítulo são aplicados tipicamente a pequenos componentes de programas (por exemplo, módulos ou pequenos grupos de módulos). O teste caixa-preta, por outro lado, amplia o seu foco e pode ser chamado de “teste no grande”.

Os testes caixa-preta são projetados para validar requisitos funcionais sem levar em conta o funcionamento interno de um programa. As técnicas caixa-preta focalizam o domínio de informações do software, derivando casos de teste e particionando o domínio de entrada e saída de um programa de forma a proporcionar uma ampla cobertura do teste. O particionamento de equivalência divide o domínio de entrada em classes de dados que tendem a usar uma função específica do software. A análise de valor limite investiga a habilidade do programa para manipular dados nos limites do aceitável. O teste de matriz ortogonal proporciona um método eficiente e sistemático para testar sistemas com poucos parâmetros de entrada. Teste baseado em modelo usa elementos do modelo de requisitos para testar o comportamento de um aplicativo.

Métodos especializados de teste abrangem um grande conjunto de recursos de software e áreas de aplicação. O teste de interfaces gráficas de usuário, arquiteturas cliente-servidor, documentação e recursos de ajuda e sistemas em tempo real requerem, cada um deles, diretrizes e técnicas especializadas.

Desenvolvedores de software experientes muitas vezes afirmam: “O teste nunca termina, ele apenas se transfere de você [o engenheiro de software] para o seu cliente. Toda vez em que um cliente usa o programa, um teste está sendo realizado”. Aplicando o projeto de caso de teste, você pode obter um teste mais completo e assim descobrir e corrigir o maior número de erros antes que comecem os “testes do cliente”.

PROBLEMAS E PONTOS A PONDERAR

18.1. Myers [Mye79] usa o seguinte programa como uma autoavaliação para a sua habilidade em especificar teste adequado: Um programa lê três valores inteiros. Os três valores são interpretados como representando os comprimentos dos lados de um triângulo. O programa imprime uma mensagem que diz se o triângulo é escaleno, isósceles, ou equilátero. Desenvolva um conjunto de casos de teste que você acha que testará adequadamente esse programa.

18.2. Projete e implemente o programa (com manipulação de erro onde for apropriado) especificado no Problema 18.1. Crie um grafo de fluxo para o programa e aplique teste de caminho base para desenvolver casos de teste que garantirão que todos os comandos no programa foram testados. Execute os casos e mostre os seus resultados.

18.3. Você consegue pensar em objetivos de teste adicionais que não foram discutidos na Seção 18.1.1?

18.4. Selecione um componente de software que você tenha projetado e implementado recentemente. Projete um conjunto de casos de teste que garantirão que todos os comandos foram executados usando teste de caminho base.

18.5. Especifique, projete e implemente uma ferramenta de software que calculará a complexidade ciclomática para a linguagem de programação de sua escolha. Use uma matriz de grafo como estrutura de dados operativos em seu projeto.

18.6. Leia Beizer [Bei95] ou alguma outra fonte de informação relacionada com a Internet (por exemplo, www.laynetworks.com/Discrete%20Mathematics_1g.htm) e determine como o programa que você desenvolveu no Problema 18.5 pode ser ampliado para acomodar vários pesos de ligação. Amplie a sua ferramenta para probabilidades de execução de processo ou tempos de processamento de ligação.

18.7. Projete uma ferramenta automatizada que reconheça ciclos e os classifique conforme indicado na Seção 18.5.3.

18.8. Amplie a ferramenta descrita no Problema 18.7 para gerar casos de teste para cada categoria de ciclo, quando encontrado. Será necessário executar essa função interativamente com o testador.

18.9. Dê pelo menos três exemplos nos quais o teste caixa-preta pode dar a impressão de que “está tudo OK”, enquanto os testes caixa-branca podem descobrir um erro. Dê pelo menos três exemplos nos quais o teste caixa-branca pode dar a impressão de que “está tudo OK”, enquanto os testes caixa-preta podem descobrir um erro.

18.10. Poderá o teste exaustivo (mesmo que ele seja possível para programas muito pequenos) garantir que o programa está 100% correto?

18.11. Teste um manual de usuário (ou um recurso de ajuda) para uma aplicação que você usa frequentemente. Encontre pelo menos um erro na documentação.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Praticamente todos os livros dedicados a teste de software consideram tanto estratégia quanto táticas. Portanto, as leituras complementares citadas no Capítulo 17 são igualmente aplicáveis para este capítulo. Everett e Raymond (*Software Testing*, Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing*, Wiley, 2007), Spiller et al. (*Software Testing Process: Test Management*, Rocky Nook, 2007), Perry (*Effective Methods for Software Testing*, 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement*, 2d ed., Auerbach, 2004), Loveland et al. (*Software Testing Techniques*, Charles River Media, 2004), Burnstein (*Practical Software Testing*, Springer, 2003), Dustin (*Effective Software Testing*, Addison-Wesley, 2002), Craig e Kaskiel (*Systematic Software Testing*, Artech House, 2002), Tamres (*Introducing Software Testing*, Addison-Wesley, 2002) e Whittaker (*How to Break Software*, Addison-Wesley, 2002) são apenas uma pequena amostra dos muitos livros que discutem princípios de teste, conceitos, estratégias e métodos.

Uma segunda edição do trabalho clássico de Myers [Mye79] foi produzida por Myers e seus colegas (*The Art of Software Testing*, 2d ed., Wiley, 2004) e abrange técnicas de projeto de caso de teste com detalhe considerável. Pezze e Young (*Software Testing and Analysis*, Wiley, 2007), Perry (*Effective Methods for Software Testing*, 3d ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design*, Artech, 2003), Hutcheson (*Software Testing Fundamentals*, Wiley, 2003), Jorgensen (*Software Testing: A Craftsman's Approach*, 2d ed., CRC Press, 2002) todos eles fornecem apresentações úteis de métodos e técnicas de projeto de casos de teste. A obra clássica de Beizer [Bei90] contém uma abrangência clara de técnicas caixa-branca, introduzindo um nível de rigor matemático que frequentemente falta em outros tratados sobre teste. Seu mais recente livro [Bei95] apresenta um tratamento conciso de importantes métodos.

Teste de software é uma atividade intensiva em termos de recursos. É por essa razão que muitas organizações automatizam partes do processo de teste. Livros de Li e Wu (*Effective Software Test Automation*, Sybex, 2004); Mosely e Posey (*Just Enough Software Test Automation*, Prentice-Hall, 2002); Dustin, Rashka, e Poston (*Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999); Graham et al. (*Software Test Automation*, Addison-Wesley, 1999); e Poston (*Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) discutem ferramentas, estratégias e métodos para teste automático. Nguyen et al. (*Global Software Test Automation*, Happy About Press, 2006) apresentam uma visão geral executiva da automação de teste.

Thomas e seus colegas (*Java Testing Patterns*, Wiley, 2004) e Binder [Bin99] descrevem padrões de teste que abrangem métodos de teste, classes/clusters, subsistemas, componentes reutilizáveis, frameworks e sistemas, bem como automação de teste e testes especializados de bases de dados.

Há disponível na Internet uma ampla variedade de recursos de informação sobre métodos de projeto de casos de teste. Uma lista atualizada das referências relevantes para as técnicas de teste pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

TESTANDO APLICAÇÕES ORIENTADAS A OBJETO

CONCEITOS-CHAVE

teste aleatório ...	462
sequência e execução (thread)	458
teste baseado em cenário	460
teste baseado em falhas	459
teste baseado em uso	458
teste de classe ...	457
teste de conjunto ...	458
teste de múltiplas classes	464
teste de partição ...	463

No Capítulo 18, vimos que o objetivo do teste é encontrar o maior número possível de erros com um trabalho razoável durante um intervalo de tempo real. Embora esse objetivo fundamental permaneça inalterado, para software orientado a objeto (*object-oriented*, OO), a natureza dos programas orientados a objeto muda tanto a estratégia de teste quanto suas táticas.

Poderíamos argumentar que, como o tamanho das bibliotecas de classe reutilizáveis aumenta, uma maior reutilização dispensará a necessidade de um teste mais pesado dos sistemas orientados a objeto. Exatamente o oposto é verdadeiro. Binder [Bin94b] discute isso quando diz:

Cada reutilização é um novo contexto de uso e é prudente o reteste. Parece provável que mais testes sejam necessários e não menos, para se atingir uma alta confiabilidade em sistemas orientados a objeto.

Para testar adequadamente os sistemas orientados a objeto, três coisas precisam ser feitas: (1) a definição do teste deve ser ampliada para incluir as técnicas de descoberta de erro aplicadas à análise orientada a objeto e modelos de projeto, (2) a estratégia para teste de unidade e de integração deve mudar significativamente e (3) o projeto de casos de teste deve levar em conta as características especiais do software orientado a objeto.

PANORAMA

O que é? A arquitetura do software orientado a objeto (OO) resulta em uma série de subsistemas em camadas que encapsulam classes colaboradoras.

Cada um desses elementos de sistema (subsistemas e classes) executa funções que ajudam a satisfazer os requisitos do sistema. É necessário testar um sistema orientado a objeto em diversos níveis em um esforço para descobrir erros que podem ocorrer à medida que as classes colaboram umas com as outras e os subsistemas se comunicam através das camadas da arquitetura.

Quem realiza? O teste orientado a objeto é executado por engenheiros de software e especialistas em teste.

Porque é importante? É necessário executar o programa antes que seja entregue ao cliente para remover todos os erros, de maneira que o usuário não passe por nenhuma frustração com um produto de má qualidade. Para encontrar o maior número possível de erros, devem ser feitos testes sistematicamente e projetados casos de teste usando técnicas bem definidas.

Quais são as etapas envolvidas? O teste orientado a objeto é estrategicamente análogo ao de sistemas convencionais, mas é taticamente diferente. Em virtude de os modelos de análise e projeto orientados a objeto serem similares em estru-

tura e conteúdo ao programa orientado a objeto resultante, o “teste” é iniciado com a revisão dos modelos. Uma vez gerado o código, começa o teste orientado a objeto “inicialmente” com o teste de classes. É projetada uma série de testes que exercitam as operações de classe e examinam se existem erros enquanto uma classe colabora com outras. À medida que as classes são integradas para formar um subsistema, aplicam-se testes baseados em sequências de execução (*thread*), baseados no uso e o teste de conjunto juntamente com abordagens baseadas em falhas para usar completamente as classes colaboradoras. Por fim, são usados casos de uso (desenvolvidos como parte do modelo de requisitos) para descobrir erros em nível de validação de software.

Qual é o artefato? É projetado e documentado um conjunto de casos de teste, desenvolvidos para usar as classes, suas colaborações e comportamentos; são definidos os resultados esperados e registrados os obtidos.

Como garantir que o trabalho foi realizado corretamente? Ao iniciar o teste, mude o seu ponto de vista. Tente por todos os meios “quebrar” o software! Projete casos de teste de maneira disciplinada e reveja os casos que você criou para que sejam completos.

19.1 AMPLIANDO A VERSÃO DO TESTE

A construção de software orientado a objeto inicia-se com a criação dos requisitos (análises) e projetos de modelo.¹ Devido à natureza evolucionária do paradigma da engenharia de software orientado a objeto, esses modelos começam como representações relativamente informais de requisitos de sistema e evoluem para modelos detalhados de classes, relações entre classes, projeto e alocação de sistema e projeto de objeto (incorporando um modelo da conectividade de objeto via mensagem). Em cada estágio, os modelos podem ser “testados” para descobrir erros antes de sua propagação para a próxima iteração.

Pode-se argumentar que a revisão da análise orientada a objeto e dos modelos de projeto é especialmente útil porque a mesma construção semântica (por exemplo, classes, atributos, operações, mensagens) aparece nos níveis de análise, projeto e código. A detecção de um problema na definição de atributos de classe durante a análise evitará efeitos colaterais que poderiam ocorrer se o problema não fosse descoberto antes do projeto ou codificação (ou mesmo a próxima iteração da análise).

Por exemplo, considere uma classe na qual um conjunto de atributos é definido durante a primeira iteração da análise. Um atributo estranho é acrescentado à classe (decorrente da falta de entendimento do domínio do problema). Duas operações são então especificadas para manipular o atributo. É feita uma revisão e um especialista em domínios aponta o problema. Eliminando o atributo estranho nesse estágio, problemas e trabalho desnecessário podem ser evitados durante a análise:

1. Poderiam ter sido geradas subclasses especiais para acomodar o atributo desnecessário ou suas exceções. O trabalho envolvido na criação de subclasses desnecessárias foi evitado.
2. Uma interpretação incorreta da definição de classe pode levar a relações de classe incorretas ou estranhas.
3. O comportamento do sistema ou suas classes pode ser caracterizado inadequadamente para acomodar o atributo estranho.

Se o problema não fosse descoberto (por meio de revisão antecipada) durante a análise e se propagasse, poderiam ocorrer as seguintes situações durante o projeto:

1. Pode ocorrer alocação imprópria da classe para o subsistema e/ou tarefas durante o projeto do sistema.
2. Pode ser despendido trabalho de projeto desnecessário para criar o projeto procedural para as operações que lidam com o atributo estranho.
3. O modelo de mensagens estará incorreto (porque devem ser designadas mensagens para as operações estranhas).

Se o problema não for detectado durante o projeto e passar para a atividade de codificação, será despendido um considerável esforço na geração de código que implementa um atributo desnecessário, duas operações desnecessárias, mensagens que controlam comunicação entre objetos e muitos outros problemas relacionados. Além disso, o teste da classe absorverá mais tempo do que o necessário. Uma vez descoberto o problema, deve ser feita a modificação do sistema, levando-se sempre em conta a alta possibilidade de efeitos colaterais causados pela alteração.

Durante estágios posteriores de seu desenvolvimento, os modelos de análise orientada a objeto (*object-oriented analysis*, OOA) e projeto orientado a objeto (*object-oriented design*, OOD) proporcionam informações substanciais sobre a estrutura e comportamento do sistema. Por essa razão, esses modelos deverão ser submetidos a rigorosa revisão antes da geração do código.



Embora a revisão da análise orientada a objeto e dos modelos de projeto seja parte integral do “teste” de uma aplicação orientada a objeto, lembre-se de que isso não é suficiente. Devem-se realizar testes executáveis também.

“As ferramentas que utilizamos têm uma profunda (e definitiva!) influência sobre nossos hábitos de pensar, e, portanto, em nossas habilidades de pensar.”

Edsger Dijkstra

¹ As técnicas de análise e modelagem de projeto são apresentadas na Parte 2 deste livro. Os conceitos orientados a objeto básicos são apresentados no Apêndice 2.

Todos os modelos orientados a objeto deverão ser testados (nesse contexto, o termo *teste* incorpora revisões técnicas) quanto à sua exatidão, integralidade e consistência com o contexto de sintaxe do modelo, semânticas e pragmatismo [Lin94a].

19.2 TESTANDO MODELOS DE ANÁLISE ORIENTADA A OBJETO (OOA) E PROJETO ORIENTADO A OBJETO (OOD)

Análises e modelos de projeto não podem ser testados no sentido convencional, pois não podem ser executados. No entanto, podem ser usadas revisões técnicas (Capítulo 15) para examinar sua exatidão e consistência.

19.2.1 Exatidão dos modelos de OOA e OODs

A notação e a sintaxe usadas para representar modelos de análise e projeto estará vinculada a métodos específicos de análise e projeto escolhidos para o projeto. Consequentemente a exatidão sintática é julgada com base no uso apropriado da simbologia; cada modelo é revisado para garantir que sejam mantidas as convenções apropriadas de modelagem.

Durante a análise e o projeto, pode-se verificar a exatidão semântica de acordo com o modelo no domínio do problema do mundo real. Se o modelo reflete com precisão o mundo real, (com um nível de detalhes apropriado para o estágio de desenvolvimento no qual o modelo é revisado), ele é semanticamente correto. Para determinar se de fato o modelo reflete os requisitos do mundo real, deve ser apresentado aos especialistas em domínio de problema que examinarão as definições de classe e a hierarquia quanto a omissões e ambiguidades. Relações de classe (conexões de instância) são avaliadas para determinar se refletem precisamente as conexões do objeto no mundo real.²

19.2.2 Consistência dos modelos orientados a objeto

A consistência de modelos orientados a objeto pode ser avaliada “considerando-se a relação entre entidades no modelo. Um modelo de análise ou projeto inconsistente tem representações em uma parte que não são refletidas corretamente em outras partes do modelo” [McG94].

Para avaliarmos a consistência, devemos examinar cada classe e suas conexões com as outras classes. O modelo classe-responsabilidade-colaboração (CRC) ou um diagrama de relacionamentos entre objetos pode ser usado para facilitar tal atividade. Conforme vimos no Capítulo 6, o modelo CRC é composto de cartões de índice CRC. Cada cartão lista o nome da classe, suas responsabilidades (operações) e colaboradores (outras classes às quais envia mensagens e das quais depende para a execução de suas responsabilidades). As colaborações implicam uma série de relacionamentos (conexões) entre classes do sistema orientado a objeto. O modelo de relacionamento de objeto fornece uma representação gráfica das conexões entre classes. Todas essas informações podem ser obtidas do modelo de análise (Capítulos 6 e 7).

Para avaliar o modelo de classe, recomendam-se os seguintes passos [McG94]:

1. **Rever o modelo CRC e o modelo de relacionamento de objeto.** Faça uma verificação comparativa para garantir que todas as colaborações relacionadas ao modelo de requisitos estão corretamente refletidas em ambos.
2. **Inspecionar a descrição de cada cartão de índice CRC para determinar se uma responsabilidade delegada faz parte da definição do colaborador.** Por exemplo, considere uma classe definida para um sistema de controle de ponto de venda que se chama **CréditoDeVenda**. Essa classe tem um cartão de índice CRC conforme mostra a Figura 19.1.

² Casos de uso podem ser valiosos para rastrear modelos de análise e projeto em um cenário de uso no mundo real para o sistema orientado a objeto.

FIGURA 19.1

Um exemplo de um cartão de índice CRC usado para revisão

nome da classe: créditoDeVenda	
tipo da classe: eventoDeTransação	
característica da classe: nãoTangível, atômica, sequencial, permanente, protegida	
responsabilidades:	colaboradores:
lerCartãoCrédito	cartãoDeCrédito
pegarAutorização	autoridadeDeCédito
solicitarValorDaCompra	tiqueteDoProduto
	registroDoVendedor
	arquivoDeBalanço
gerarConta	conta

Para essa coleção de classes e colaborações, pergunte se uma responsabilidade (por exemplo, *lerCartãoCrédito*) é atendida se for delegada ao colaborador citado (**CartãoDeCrédito**). Isto é, a classe **CartãoDeCrédito** tem uma operação que a habilita a ser lida? Nesse caso, a resposta é “sim”. A relação-objeto é percorrida para garantir que todas essas conexões são válidas.

- Inverta a conexão para garantir que cada colaborador ao qual é solicitado serviço esteja recebendo solicitações de uma origem razoável.** Por exemplo, se a classe **CartãoDeCrédito** receber uma solicitação para **purchase amount (valor da compra)** da classe **CréditoDeVenda**, haverá um problema. **CartãoDeCrédito** não conhece o valor da compra.
- Usando as conexões invertidas examinadas na etapa 3, determine se podem ser necessárias outras classes ou se as responsabilidades estão corretamente agrupadas entre as classes.
- Determine se responsabilidades solicitadas mais amplas podem ser combinadas em uma única responsabilidade.** Por exemplo, *lerCartãoCrédito* e *pegarAutorização* ocorrem em todas as situações. Elas podem ser combinadas em uma responsabilidade de *validação de solicitação de crédito*, que incorpora a obtenção do número do cartão de crédito e obtenção da autorização.

Você deverá aplicar os passos de 1 a 5 iterativamente a cada classe e através de cada evolução do modelo de requisitos.

Uma vez criado o modelo de projeto (Capítulos 9 a 11), você deverá realizar também revisões do projeto do sistema e projeto do objeto. O projeto do sistema representa a arquitetura geral do produto, os subsistemas que compõem o produto, a maneira pela qual os subsistemas são alocados a processadores, a alocação de classes para subsistemas e o projeto da interface do usuário. O modelo de objeto apresenta os detalhes de cada classe e as atividades de mensagens necessárias para implementar colaborações entre classes.

O projeto do sistema é revisado examinando-se o modelo objeto-comportamento desenvolvido durante a análise orientada a objeto e mapeando o comportamento requerido do sistema em relação aos subsistemas projetados para atender a esse comportamento. A concorrência e a alocação de tarefas também é revista segundo o contexto de comportamento do sistema. Os estados comportamentais do sistema são avaliados para determinar quais existem concorrentemente. São usados casos de uso para exercitar o projeto da interface do usuário.

O modelo orientado a objeto deverá ser testado em relação à rede objeto-relacionamento para assegurar que todos os objetos de projeto contenham os atributos e operações necessários para implementar as colaborações definidas para cada cartão de Índice CRC. Além disso, é revisada a especificação dos detalhes de operação (isto é, os) algoritmos que implementam as operações).

19.3 ESTRATÉGIAS DE TESTE ORIENTADO A OBJETO

Conforme mencionado no Capítulo 18, a estratégia de teste de software clássica começa com o “teste no pequeno” e se amplia para o “teste no grande”. De acordo com o jargão do teste de software (Capítulo 18), você começa com *teste de unidade*, depois passa para o *teste de integração* e termina com a *validação e teste do sistema*. Em aplicações convencionais, o teste de unidade focaliza a menor unidade de programa compilável — o subprograma (por exemplo, componente, módulo, sub-rotina, procedimento). Depois que cada uma dessas unidades é testada individualmente, elas são integradas em uma estrutura de programa enquanto é executada uma série de testes de regressão para descobrir erros devidos ao interfaceamento de módulos e os efeitos colaterais causados pela adição de novas unidades. Por fim, o sistema como um todo é testado para assegurar que sejam descobertos os erros em requisitos.

19.3.1 Teste de unidade em contexto orientado a objeto

PONTO-CHAVE

A menor “unidade” que pode ser testada em software orientado a objeto é a classe. O teste de classe é controlado pelas operações encapsuladas pela classe e o comportamento de estado da classe.

Quando se considera o software orientado a objeto, o conceito de unidade muda. O encapsulamento controla a definição de classes e objetos. Cada classe e cada instância de uma classe (objeto) empacotam (pacotes) atributos (dados) e as operações (também conhecidas como métodos ou serviços) que manipulam esses dados. Em vez de testar um módulo individual, a menor unidade testável é a classe encapsulada. Pelo fato de uma classe poder conter muitas operações diferentes e uma operação em particular poder existir como parte de um conjunto de classes diferentes, o significado do teste de unidade muda significativamente.

Já não podemos mais testar uma única operação isoladamente (a visão convencional do teste de unidade), mas como parte de uma classe. Considere uma hierarquia de classe na qual uma operação $X()$ é definida para a superclasse e é herdada por um conjunto de subclasses. Cada subclasse usa a operação $X()$, mas ela é aplicada de acordo com o contexto de atributos privados e operações definidas para cada subclasse. O contexto no qual a operação $X()$ é usada varia de maneira sutil, desse modo, é necessário testar a operação $X()$ no contexto de cada uma das subclasses. Isso significa que testar a operação $X()$ em um vácuo (a abordagem tradicional de teste de unidade) é ineficaz no contexto orientado a objeto.

O teste de classe para software orientado a objeto equivalente ao teste de unidade para software convencional.³ Ao contrário do teste de unidade para software convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem através da interface do módulo, o teste de classe para software orientado a objeto é controlado pelas operações encapsuladas pela classe e o comportamento de estado da classe.

19.3.2 Teste de integração em contexto orientado a objeto

Como o software orientado a objeto não tem uma estrutura de controle hierárquico, as estratégias de integração convencionais de cima para baixo e de baixo para cima têm pouco significado. Além disso, integrar as operações uma de cada vez em uma classe (a abordagem convencional incremental de integração) frequentemente é impossível devido às “interações direta e indireta dos componentes que formam a classe” [Ber93].

Há duas estratégias diferentes para teste de integração de sistemas orientados a objeto [Bin94a]. A primeira, *teste baseado em sequências de execução (thread-based testing)*, integra o conjunto de classes necessárias para responder a uma entrada ou evento para o sistema. Cada

3 Os métodos de projeto de casos de teste para classes orientadas a objeto são discutidos nas Seções 19.4 a 19.6.

PONTO-CHAVE

O teste de integração para software orientado a objeto testa um conjunto de classes necessárias para responder a determinado evento.

sequência de execução (*thread*) é integrada e testada individualmente. O teste de regressão é aplicado para garantir que não ocorram efeitos colaterais. A segunda abordagem de integração, *teste baseado em uso*, começa a construção do sistema testando aquelas classes (chamadas *classes independentes*) que usam bem poucas (se usar alguma) classes servidoras. Depois que as classes independentes são testadas, testa-se a próxima camada de classes, chamadas de *classes dependentes*, que usam as classes independentes. Essa sequência de teste de camadas de classes dependentes continua até que o sistema inteiro seja construído. Diferentemente da integração convencional, o uso de pseudocontrolador e pseudocontrolados (Capítulo 18), como operações substitutas, deve ser evitado quando possível.

Teste de conjunto [McG94] é uma etapa do teste de integração de software orientado a objeto. Aqui, um *conjunto* de classes colaboradoras (determinado examinando-se o CRC e o modelo de relacionamento de objeto) é exercitado projetando-se casos de teste que tentam descobrir erros nas colaborações.

19.3.3 Teste de validação em contexto orientado a objeto

Em nível de validação ou de sistema, os detalhes das conexões de classes desaparecem. Assim como a validação convencional, a validação de software orientado a objeto focaliza as ações visíveis pelo usuário e as saídas do sistema reconhecíveis pelo usuário. Para ajudar na criação de testes de validação, o testador deverá fundamentar-se nos casos de uso (Capítulos 5 e 6) que fazem parte do modelo de requisitos. O caso de uso proporciona um cenário com grande possibilidade de detectar erros em requisitos de interação de usuário.

Os métodos convencionais de teste caixa-preta (Capítulo 18) podem ser usados para controlar testes de validação. Além disso, pode-se optar por criar casos de teste com base no modelo de comportamento de objeto e em um diagrama de fluxo de evento criado como parte da análise orientada a objeto.

19.4 MÉTODOS DE TESTE ORIENTADOS A OBJETO

“Encaro os testadores como os guarda-costas do projeto. Nós defendemos o flanco de nossos desenvolvedores contra as falhas, enquanto eles concentram-se na efetivação do sucesso.”

James Bach

A arquitetura de software orientado a objeto resulta em uma série de subsistemas em camada que encapsulam as classes de colaboração. Cada um desses elementos de sistema (subsistemas e classes) executa funções que ajudam a satisfazer os requisitos do sistema. É necessário testar um sistema orientado a objeto em uma variedade de níveis diferentes para descobrir erros que podem ocorrer à medida que as classes colaboram umas com as outras e os subsistemas se comunicam através de camadas da arquitetura.

Os métodos de projeto de casos de teste para software orientado a objeto continuam a evoluir. No entanto, uma abordagem geral para projeto de casos de teste orientado a objeto foi sugerida por Berard [Ber93]:

1. Cada caso de teste deverá ser identificado de forma única e associado explicitamente com a classe a ser testada.
2. Deverá ser definida a finalidade do teste.
3. Deverá ser feita uma lista das etapas de teste para cada teste e ela deverá conter:
 - a. Uma lista dos estados especificados para a classe a ser testada
 - b. Uma lista das mensagens e operações que serão executadas em consequência do teste
 - c. Uma lista das exceções que podem ocorrer enquanto a classe é testada
 - d. Uma lista de condições externas (alterações no ambiente externo ao software que deve existir para fazer corretamente o teste)
 - e. Informações suplementares que ajudarão a entender ou implementar o teste

Diferentemente do projeto convencional de casos de teste, controlado por uma visão entrada-processo-saída do software ou do detalhe algorítmico dos módulos individuais, o teste

orientado a objeto concentra-se no planejamento de sequências apropriadas de operação para executar os estados de uma classe.

19.4.1 As implicações no projeto de casos de teste dos conceitos orientados a objeto

WebRef

Uma excelente coleção de publicações e recursos sobre teste orientado a objeto pode ser encontrada em www.rbsc.com.

À medida que uma classe evolui através dos modelos de requisitos e de projeto, torna-se alvo para o projeto de casos de teste. Como os atributos e operações são encapsulados, as operações de teste fora da classe em geral são improdutivas. Embora o encapsulamento seja um conceito de projeto essencial para software orientado a objeto, ele pode criar um pequeno obstáculo na realização do teste. Conforme afirma Binder [Bin94a], “O teste requer um relato sobre o estado concreto e abstrato de um objeto”. Ainda assim, o encapsulamento pode dificultar a obtenção dessa informação. A menos que sejam inseridas operações internas para relatar os valores dos atributos da classe, um resumo do estado de um objeto pode ser difícil de obter.

A herança também pode apresentar desafios adicionais durante o projeto de casos de teste. Observamos que a cada novo contexto de uso é requerido um reteste, mesmo que tenha sido possível a reutilização. Além disso, a herança múltipla⁴ complica ainda mais o teste, aumentando o número de contextos para os quais é necessário o teste [Bin94a]. Se subclasses instanciadas a partir de uma superclasse forem usadas dentro do mesmo domínio do problema, é provável que o conjunto de casos de teste derivado para a superclasse possa ser usado para testar a subclasse. No entanto, se a subclasse for usada em um contexto inteiramente diferente, os casos de teste da superclasse terão pouca aplicabilidade, e um novo conjunto de testes precisa ser projetado.

19.4.2 Aplicabilidade dos métodos convencionais de projeto de casos de teste

Os métodos de teste caixa-branca descritos no Capítulo 18 podem ser aplicados às operações definidas para uma classe. Técnicas de caminho-base, teste de ciclos ou fluxo de dados podem ajudar a garantir que cada instrução em uma operação tenha sido testada. No entanto, a estrutura concisa de muitas operações de classe sugere argumentos de que o esforço aplicado no teste caixa-branca poderia ser mais bem redirecionado para testes no nível de classe.

Os métodos de teste caixa-preta são apropriados para sistemas orientados a objeto, assim como para sistemas desenvolvidos por meio de métodos convencionais de engenharia de software. Conforme mencionado no Capítulo 18, casos de uso podem proporcionar informações úteis no projeto de testes caixa-preta e baseados em estado.

19.4.3 Teste baseado em falhas⁵

PONTO-CHAVE

A estratégia para o teste baseado em falhas é formular uma hipótese de uma série de falhas possíveis e criar testes para provar cada uma das hipóteses.

O objetivo do *teste baseado em falhas* em um sistema orientado a objeto é projetar testes que tenham grande probabilidade de descobrir falhas plausíveis. Como o produto ou sistema deve satisfazer os requisitos do cliente, o planejamento preliminar necessário para realizar o teste baseado em falhas começa com o modelo de análise. O testador procura por falhas plausíveis (aspectos da implementação do sistema que podem resultar em defeitos). Para determinar se essas falhas existem, projetam-se casos de teste para exercitar o projeto ou código.

Naturalmente, a eficácia dessas técnicas depende de como os testadores consideram uma falha plausível. Se falhas reais em um sistema orientado a objeto são vistas como não plausíveis, essa abordagem não é melhor do que qualquer técnica de teste aleatório. No entanto, se os modelos de análise e projeto puderem proporcionar conhecimento aprofundado sobre o que provavelmente pode sair errado, o teste baseado em falhas pode encontrar quantidade significativa de erros com esforço relativamente pequeno.

⁴ Um conceito orientado a objeto que deverá ser usado com extremo cuidado.

⁵ As Seções 19.4.3 a 19.4.6 foram adaptadas de um artigo de Brian Marick postado na Internet no newsgroup comp.testing. Essa adaptação foi incluída com permissão do autor. Para mais informações sobre esses tópicos, veja [Mar94]. Deve-se notar que as técnicas discutidas nas Seções 19.4.3 a 19.4.6 são também aplicáveis para software convencional.

? Que tipos de falhas são encontrados em chamadas de operação e nas conexões de mensagens?

O teste de integração procura por falhas plausíveis em chamadas de operação ou conexões de mensagem. Três tipos de falhas encontram-se nesse contexto: resultado inesperado, uso de operação/mensagem errada e invocação incorreta. Para determinar as falhas plausíveis quando funções (operações) são invocadas, o comportamento da operação deve ser examinado.

O teste de integração se aplica tanto a atributos quanto a operações. Os “comportamentos” de um objeto são definidos pelos valores atribuídos a seus atributos. O teste deve exercitar os atributos para determinar se ocorrem os valores apropriados para tipos distintos de comportamento de objeto.

É importante notar que o teste de integração tenta encontrar erros no objeto-cliente, não no servidor. Em termos convencionais, o foco do teste de integração é determinar se existem erros no código chamador, não no código chamado. A chamada da operação é usada como um indício, uma maneira de encontrar os requisitos de teste que usam o código chamador.

19.4.4 Casos de teste e a hierarquia de classe

A herança não torna óbvia a necessidade de um teste completo de todas as classes derivadas. De fato, ela pode realmente complicar o processo de teste. Considere a seguinte situação. Uma classe **Base** contém operações *herdada()* e *redefinida()*. Uma classe **Derivada** redefine *redefinida()* para servir em um contexto local. Há pouca dúvida de que **Derivada::redefinida()** tem de ser testada porque representa um novo projeto e um novo código. Mas **Derivada::herdada()** tem de ser retestada?

Se **Derivada::herdada()** chama *redefinida()* e o comportamento de *redefinida()* foi alterado, **Derivada::herdada()** pode processar de maneira errada o novo comportamento. Portanto, precisa de novos testes apesar de o projeto e código não terem mudado. É importante notar que apenas um subconjunto de todos os testes para **Derivada::herdada()** pode ter de ser realizado. Se parte do projeto e código para *herdada()* não depende de *redefinida()* (não o chama nem chama qualquer código que o chame indiretamente), aquele código não precisa ser retestado na classe derivada.

Base::redefinida() e **Derivada::redefinida()** são duas operações diferentes com diferentes especificações e implementações. Cada uma teria um conjunto de requisitos de teste derivados da especificação e implementação. Estes investigam falhas plausíveis: falhas na integração, falhas de condição, falhas de limites e assim por diante. Mas as operações provavelmente serão similares. Seus conjuntos de requisitos de teste vão se sobrepor. Quanto melhor for o projeto orientado a objeto, maior é a sobreposição. Precisam ser obtidos novos testes somente para aqueles requisitos **Derivada::redefinida()** que não são satisfeitos pelos testes **Base::redefinida()**.

Para resumir, os testes **Base::redefinida()** são aplicados a objetos da classe **Derivada**. Entradas de teste podem ser apropriadas para as classes base e derivada, mas os resultados esperados podem ser diferentes na classe derivada.

19.4.5 Projeto de teste baseado em cenário

O teste baseado em falhas omite dois tipos principais de erro: (1) especificações incorretas e (2) interações entre subsistemas. Quando ocorrem erros associados a uma especificação incorreta, o produto não realiza o que o cliente deseja. Pode fazer alguma coisa errada ou omitir uma funcionalidade importante. Mas em qualquer circunstância, a qualidade (conformidade com os requisitos) é prejudicada. Erros associados com interação de subsistema ocorrem quando o comportamento de um subsistema cria circunstâncias (por exemplo, eventos, fluxo de dados) que faz um outro subsistema falhar.

O teste baseado em cenário concentra-se naquilo que o usuário faz, não no que o produto faz. Isso significa detectar as tarefas (por meio de casos de uso) que o usuário tem de executar e aplicá-las, bem como suas variantes como testes.

PONTO-CHAVE

Mesmo que uma classe base tenha sido completamente testada, será preciso testar todas as classes derivadas dela.

PONTO-CHAVE

Testes baseados em cenário indicarão erros que ocorrem quando qualquer ator interage com o software.

Cenários descobrem erros de interação. Mas, para tanto, os casos de teste devem ser mais complexos e realistas do que os testes baseados em falhas. Os testes baseados em cenário tendem a usar múltiplos subsistemas em um único teste (os usuários não se limitam ao uso de um subsistema de cada vez).

Como exemplo, considere o projeto de testes baseados em cenário para um editor de texto. Os casos de uso são apresentados a seguir:

Caso de uso: corrigir o esboço final

Background: é comum imprimir o esboço “final”, ler e descobrir alguns erros evidentes que não estavam óbvios na tela. Esse caso de uso descreve a sequência de eventos que ocorre quando isso acontece.

1. Imprima o documento inteiro.
2. Reveja o documento alterando algumas páginas.
3. À medida que cada página é alterada, ela é impressa.
4. Algumas vezes uma série de páginas é impressa.

“Se você quer e espera que um programa funcione, provavelmente verá um programa funcionando — não verá as falhas.”

Cem Kaner et al.

Esse cenário descreve duas coisas: o teste e necessidades específicas de usuário. As necessidades do usuário são óbvias: (1) um método para imprimir páginas separadas e (2) um método para imprimir um intervalo de páginas. Do ponto de vista do teste, há necessidade de testar a edição após a impressão (e vice-versa). Portanto, o testador trabalha no projeto de testes que descobrirão erros na função de edição causados pela função de impressão; erros que indicarão que as duas funções do software não são propriamente independentes.

Caso de uso: imprimir uma nova cópia

Background: alguém solicita ao usuário uma nova cópia do documento. Ela deve ser impressa.

1. Abra o documento.
2. Imprima-o.
3. Feche o documento.

Novamente, a abordagem do teste é relativamente óbvia. Exceto que esse documento não veio de lugar nenhum. Foi criado em uma tarefa anterior. Essa tarefa afeta a presente?

Em muitos editores modernos, os documentos “lembram-se” de como foram impressos da última vez. Por padrão, são impressos da mesma maneira na próxima vez. Após o cenário **Corrigir o esboço final**, basta selecionar “Imprimir” no menu e clicar o botão Imprimir na caixa de diálogo e será impressa novamente a última página corrigida. De acordo com o editor, o cenário correto deverá se parecer com este:

Caso de uso: imprimir uma nova cópia

1. Abra o documento.
2. Selecione “Imprimir” no menu.
3. Verifique se está imprimindo um intervalo de páginas; se estiver, clique para imprimir o documento inteiro.
4. Clique o botão Imprimir.
5. Feche o documento.

Esse cenário indica um erro potencial de especificação. O editor não faz aquilo que o usuário espera que ele faça. Os clientes muitas vezes não observam a verificação da etapa 3. Ficarão irritados quando encontrarem uma página na impressora quando esperavam encontrar 100. Clientes irritados indicam problemas de especificação.

Um projetista de casos de teste poderia não perceber essa dependência quando projeta os testes, mas é possível que o problema apareça durante o teste. O testador teria de argumentar com a provável resposta, “Deve ser dessa maneira que funciona!”.



Embora o teste baseado em cenário tenha seus méritos, você terá um melhor retorno do tempo investido revisando casos de uso quando são desenvolvidos como parte do modelo de análise.

19.4.6 Teste da estrutura superficial e estrutura profunda

Estrutura superficial refere-se à estrutura observável externamente de um programa orientado a objeto; a estrutura imediatamente óbvia para um usuário final. Em vez de executar funções, os usuários de muitos sistemas orientados a objeto podem receber objetos para manipular de alguma forma. Qualquer que seja a interface, os testes ainda são baseados nas tarefas do usuário. A captura dessas tarefas envolve o entendimento, observação e diálogo com os usuários representativos (e os usuários não representativos que puderem ser encontrados).

Certamente haverá alguma diferença no detalhe. Por exemplo, em um sistema convencional com uma interface orientada a comandos, o usuário pode utilizar a lista de todos os comandos como uma verificação do teste. Se não existiam cenários de teste para simular um comando, provavelmente o teste não considerou algumas tarefas do usuário (ou a interface tem comandos inúteis). Em uma interface baseada em objeto, o testador pode usar a lista de todos os objetos como uma verificação de teste.

Os melhores testes são criados quando o projetista encara o sistema de maneira nova ou não convencional. Por exemplo, se o sistema ou produto tem uma interface baseada em comandos, serão desenvolvidos testes mais completos se o projetista do caso de teste simular as operações independentes dos objetos. Faça perguntas do tipo, “Será que o usuário vai querer utilizar essa operação — que se aplica somente ao objeto **Scanner** — enquanto estiver trabalhando com a impressora?”. Qualquer que seja o estilo da interface, o projeto do caso de teste que simula a estrutura superficial deve usar tanto objetos quanto as operações como indícios que levam a tarefas não consideradas.

A *Estrutura profunda* refere-se aos detalhes técnicos internos de um programa orientado a objeto, isto é, a estrutura entendida examinando-se o projeto e/ou código. O teste de estrutura profunda é projetado para simular dependências, comportamentos e mecanismos de comunicação estabelecidos como parte do modelo de projeto para software orientado a objeto.

Os requisitos e os modelos de teste são usados como base para o teste de estrutura profunda. Por exemplo, o diagrama de colaboração UML ou o modelo de distribuição representa colaborações entre objetos e subsistemas que podem não ser externamente visíveis. O projeto de casos de teste então pergunta: “Nós (como teste) capturamos alguma tarefa que usa a colaboração representada no diagrama de colaboração? Se não, por que não?”.

PONTO-CHAVE

O teste da estrutura superficial é análogo ao teste caixa-preta. O teste de estrutura profunda é similar ao teste caixa-branca.

“Não se envergonhe de seus erros e não os transforme em crimes.”

Confúcio

19.5 MÉTODOS DE TESTE APLICÁVEIS NO NÍVEL DE CLASSE



O número de permutações possíveis para teste aleatório pode crescer muito. Uma estratégia similar ao teste de matriz ortogonal pode ser usada para melhorar a eficiência de teste.

O teste “no pequeno” focaliza uma única classe e os métodos encapsulados pela classe. Teste aleatório e particionamento são métodos que podem ser usados para simular uma classe durante o teste orientado a objeto.

19.5.1 Teste aleatório para classes orientadas a objeto

Para apresentar breves ilustrações desses métodos, considere uma aplicação bancária na qual uma classe **Conta** tem as seguintes operações: *abrir()*, *estabelecer()*, *depositar()*, *retirar()*, *obterSaldo()*, *resumir()*, *limiteDeCrédito()* e *fechar()* [Kir94]. Cada uma dessas operações pode ser aplicada para **Conta**, mas certas restrições (por exemplo, primeiro a conta precisa ser aberta para que as outras operações possam ser aplicadas e fechada depois que todas as operações são completadas) são implícitas à natureza do problema. Mesmo com essas restrições, há muitas permutações das operações. O histórico de comportamento mínimo de uma instância de **Conta** inclui as seguintes operações:

abrir • estabelecer • depositar • retirar • fechar

Isso representa a sequência mínima de teste para **Conta**. No entanto, pode ocorrer uma ampla variedade de outros comportamentos nessa sequência:

abrir • estabelecer • depositar • [depositar | retirar] | obterSaldo | resumir | limiteDeCrédito]ⁿ • retirar • fechar

Uma variedade de diferentes sequências de operações pode ser gerada aleatoriamente. Por exemplo:

Caso de teste r_1 : **abrir**•**estabelecer**•**depositar**•**depositar**• **obterSaldo** •**resumir**•**retirar**•**fechar**

Caso de teste r_2 : **abrir**•**estabelecer**•**depositar**•**retirar**•**depositar**• **obterSaldo**•**limiteDeCrédito**•**retirar**•**fechar**

Esses e outros testes de ordem aleatória podem ser usados para exercitar diferentes históricos de duração de instância de classe.

CASA SEGURA



Teste de classe

Cena: Escritório da Shakira.

Atores: Jamie e Shakira — membros da equipe de engenharia de software que estão trabalhando no projeto de um caso de teste para função de segurança do *CasaSegura*.

Conversa:

Shakira: Desenvolvi alguns testes para a classe *Detector* [Figura 10.4] — você sabe, aquela que permite acesso a todos os objetos **Sensor** para a função de segurança. Entendeu?

Jamie (rindo): Claro, é aquela que você usou para acrescentar o sensor “raiva de cachorro”.

Shakira: Essa mesma. Bem, ela tem uma interface com quatro operações: *ler()*, *habilitar()*, *desabilitar()*, e *testar()*. Para que um sensor possa ser lido, tem de ser primeiro habilitado. Uma vez habilitado, pode ser lido e testado. Ele pode ser desabilitado a qualquer instante, exceto se uma condição de alarme estiver sendo processada. Assim, eu defini uma sequência simples de teste que irá simular sua história comportamental. [Mostra a Jamie a seguinte sequência.]

#1: **habilitar**•**testar**•**ler**•**desabilitar**

Jamie: Vai funcionar, mas você terá que fazer mais testes do que isso

Shakira: Eu sei, eu sei, aqui estão algumas outras sequências que eu descobri. [Mostra a Jamie as seguintes sequências.]

#2: **habilitar**•**testar***[**ler**]ⁿ•**testar**•**desabilitar**

#3: [**ler**]ⁿ

#4: **habilitar*****desabilitar**•[**testar** | **ler**]

Jamie: Bem, deixe-me ver se entendi o objetivo dessas sequências.

#1 acontece de maneira trivial, assim como um uso convencional.

#2 repete a operação de leitura *n* vezes, e esse é um cenário provável.

#3 tenta ler o sensor antes de ele ser habilitado... Isso deve produzir uma mensagem de erro de algum tipo, certo?

#4 habilita e desabilita o sensor e então tenta lê-lo. Isso não é o mesmo que o teste #2?

Shakira: Na verdade, não. Em #4, o sensor foi habilitado. O que #4 realmente testa é se a operação de desabilitar funciona como deveria. Um *ler()* ou *testar()* após *desabilitar()* deverá gerar uma mensagem de erro. Se isso não acontecer, temos um erro na operação de desabilitar.

Jamie: Certo. Lembre-se de que os quatro testes têm de ser aplicados a cada tipo de sensor, já que todas as operações podem ser sutilmente diferentes dependendo do tipo de sensor.

Shakira: Não se preocupe. Está planejado.

19.5.2 Teste de partição em nível de classe

Que opções de teste estão disponíveis em nível de classe?

O teste de partição reduz o número de casos de teste necessários para simular a classe de maneira muito semelhante ao particionamento de equivalência (Capítulo 18) para software tradicional. As entradas e saídas são classificadas e os casos de teste projetados para exercitar cada categoria. Mas como são criadas as categorias de particionamento?

O particionamento baseado em estado classifica as operações de classe com base na habilidade delas para mudar o estado da classe. Considerando novamente a classe **Conta**, as operações de estado incluem *depósito()* e *retirada()*, enquanto as operações de não estado incluem *saldo()*, *resumo()*, e *limiteDeCredito()*. Os testes são projetados de forma que simulam operações que mudam e que não mudam o estado, separadamente. Portanto,

Caso de teste p_1 : **abrir**•**estabelecer**•**depósito**• **depositar**•**retirar**• **retirar**•**fechar**

Caso de teste p_2 : **abrir**•**estabelecer**•**depositar**•**resumir**•**limiteDeCrédito**•**retirar**•**fechar**

O caso de teste p_1 muda o estado, enquanto o caso de teste p_2 simula operações que não mudam o estado (exceto aquelas na sequência de teste mínima).

Particionamento baseado em atributo classifica operações de classe com base nos atributos que elas usam. Para a classe **Conta**, os atributos **balance** e **limiteDeCrédito** podem ser usados para definir partições. As operações são divididas em três partições: (1) operações que usam **limiteDeCrédito**, (2) operações que modificam **limiteDeCrédito**, e (3) operações que não usam nem modificam **limiteDeCrédito**. São então projetadas sequências de teste para cada partição.

Particionamento baseado em categoria classifica operações de classe com base na função genérica que cada uma executa. Por exemplo, operações na classe **Conta** podem ser classificadas em operações de inicialização (*abrir, estabelecer*), operações computacionais (*depositar, retirar*), consultas (*obterSaldo, resumir, limiteDeCrédito*), e operações de terminação (*fechar*).

19.6 PROJETO DE CASO DE TESTE INTERCLASSE

“A fronteira que define o escopo do teste de unidade e teste de integração é diferente para o desenvolvimento orientado a objeto. Podem ser projetados e usados testes em muitos pontos no processo. Assim, ‘projetar pequeno, codificar pequeno’ torna-se ‘projetar pequeno, codificar pequeno, testar pequeno’.”

Robert Binder

O projeto de caso de teste torna-se mais complicado quando começa a integração do sistema orientado a objeto. É nesse estágio que o teste de colaborações entre classes deve começar. Para ilustrarmos a “geração de caso de teste interclasse” [Kir94], expandiremos o exemplo bancário introduzido na Seção 19.5 para incluir as classes e colaborações da Figura 19.2. As direções das setas na figura indicam a direção das mensagens, e os rótulos, as operações chamadas como consequência das colaborações sugeridas pelas mensagens.

Como o teste de classes individuais, o teste de colaboração entre classes pode ser feito aplicando-se métodos aleatórios e de particionamento, bem como teste baseado em cenário e teste comportamental.

19.6.1 Teste de múltiplas classes

Kirani e Tsai [Kir94] sugerem a seguinte sequência de passos para gerar casos de teste aleatórios de múltiplas classes:

1. Para cada classe cliente, use a lista de operações de classe para gerar uma série de sequências aleatórias de teste. As operações enviarão mensagens para outras classes servidoras.
2. Para cada mensagem gerada, determine a classe colaboradora e a operação correspondente no objeto servidor.
3. Para cada operação no objeto servidor (que foi chamado por mensagens enviadas pelo objeto cliente), determine as mensagens que ele transmite.
4. Para cada uma das mensagens, determine o próximo nível de operações chamadas e incorpore-as na sequência de teste.

Para ilustrar [Kir94], considere uma sequência de operações para a classe **Banco** relativas a uma classe **ATM** (Figura 19.2):

verificarConta • verificarPIN • [[verificarDiretrizes • requisiçãoDeRetirada] | requisiçãoDeDepósito | requisiçãoDeInformaçãoDeConta]

Um caso de teste aleatório para a classe **Banco** poderia ser

Caso de teste r_3 = **verificarConta • verificarPIN • requisiçãoDeDepósito**

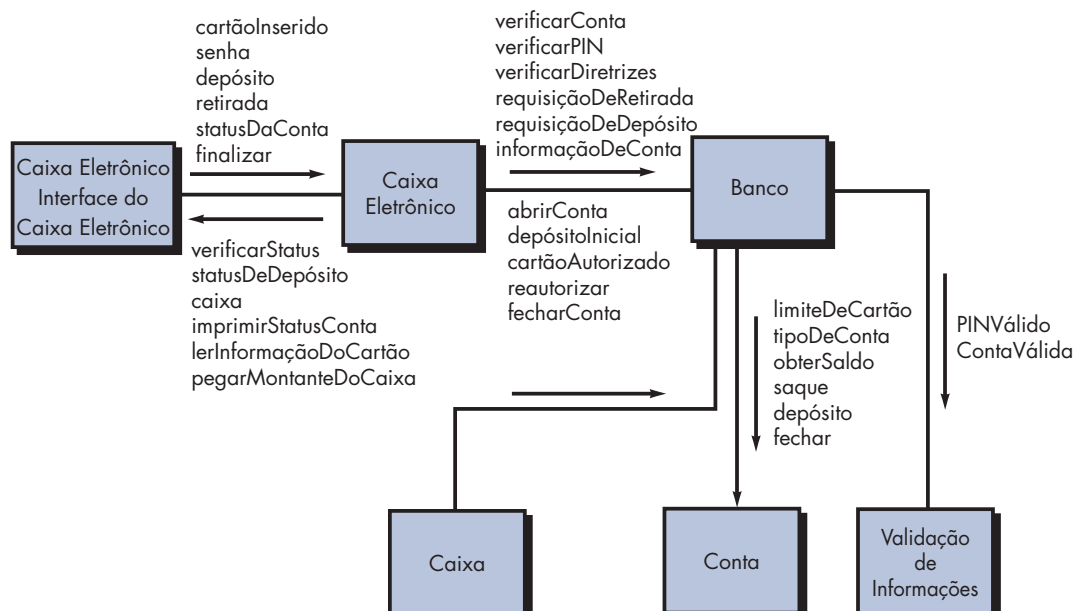
Para as colaborações envolvidas nesse teste, são consideradas as mensagens associadas a cada uma das operações citadas no caso de teste r_3 . **Banco** deve colaborar com **InformaçãoDeValidação** para executar a **verificarConta()** e **verificarPIN()**. **Banco** deve colaborar com **Conta** para executar **requisiçãoDeDepósito()**. Daí, um novo caso de teste que exercita essas colaborações é

Caso de teste r_4 = **verificarConta [Banco:contaVálidaInformaçãoDeValidação] • verificarPIN [Banco: PINVálidoInformaçãoDeValidação] • requisiçãoDeDepósito [Banco: depósitoConta]**

FIGURA 19.2

Diagrama de colaboração de classe para a aplicação bancária

Fonte: Adaptado de (Kir94)



A abordagem para teste de partição de múltiplas classes é similar à usada para teste de partição de classes individuais. Uma classe simples é particionada conforme discutido na Seção 19.5.2. No entanto, a equivalência de teste é expandida para incluir operações chamadas via mensagens a classes colaboradoras. Uma abordagem alternativa particiona os testes com base nas interfaces para uma classe em particular. Conforme a Figura 19.2, a classe **Banco** recebe mensagens das classes **ATM** e **Caixa**. Os métodos da classe **Banco** podem, portanto, ser testados particionando-os naqueles que servem **ATM** e naqueles que servem **Caixa**. Pode ser usada a partição baseada em estado (Seção 19.5.2) para refinar ainda mais as partições.

19.6.2 Testes derivados de modelos comportamentais

O uso dos diagramas de estado como um modelo que representa o comportamento dinâmico de uma classe é discutido no Capítulo 7. O diagrama de estado para uma classe pode ser usado para ajudar a derivar uma sequência de testes que irão simular o comportamento dinâmico da classe (e as classes que colaboram com ela). A Figura 19.3 [Kir94] ilustra um diagrama de estado para a classe **Conta** discutida anteriormente. Observando a figura, as transações iniciais movem-se para os estados *conta vazia* e *conta estabelecida*. A maior parte do comportamento para instâncias da classe ocorre quando ainda no estado *conta ativa*. Uma retirada final e fechamento da conta fazem a classe conta transitar para os estados *conta inativa* e *conta "morta"*, respectivamente.

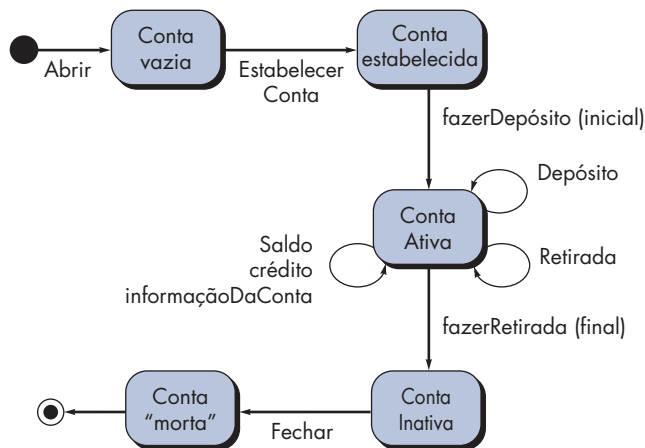
Os testes a ser projetados deverão conseguir a cobertura de todos os estados. Isto é, as sequências de operação deverão fazer a classe **Conta** realizar transição através de todos os estados permitidos:

Caso de teste s_1 : **abrir • estabelecerConta • fazerDepósito (inicial) • fazerRetirada (final) • fechar**

Deve-se notar que essa sequência é idêntica à sequência de teste mínimo discutida na Seção 19.5.2. Acrescentando sequências de teste adicionais à sequência mínima,

Caso de teste s_2 : **abrir • estabelecerConta • fazerDepósito (inicial) • fazerDepósito • obterSaldo • obter LimiteDeCrédito • fazerRetirada (final) • fechar**

Caso de teste s_3 : **abrir • estabelecerConta • fazerDepósito(inicial) • fazerDepósito • fazerRetirada • informaçãoDaConta • fazerRetirada (final) • fechar**

FIGURA 19.3**Diagrama de estado para a classe Conta****Fonte:** Adaptado de (Kir94)

Mais casos de teste poderiam ainda ser criados para garantir que todos os comportamentos para a classe fossem adequadamente simulados. Em situações nas quais o comportamento da classe resulta em uma colaboração com uma ou mais classes, são usados diagramas de estados múltiplos para acompanhar o fluxo comportamental do sistema.

O modelo de estado pode ser percorrido de uma maneira primeiro-em-largura [McG94]. Nesse contexto, primeiro-em-largura implica que um caso de teste simula uma única transição e que, quando uma nova transição deve ser testada, somente as transições testadas anteriormente são usadas.

Considere um objeto **CartãoDeCrédito** que faz parte do sistema bancário. O estado inicial de **CartãoDeCrédito** é *indefinido* (não foi fornecido nenhum número de cartão de crédito). Após ler o cartão de crédito durante uma venda, o objeto assume um estado *definido*; isto é, os atributos **card number** e **expiration date**, juntamente com identificadores específicos do banco são definidos. O cartão de crédito é *submetido* (enviado) quando ele é enviado para autorização, e é aprovado quando a autorização é recebida. A transição de **CartãoDeCrédito** de um estado para outro pode ser testada derivando casos de teste que fazem a transição ocorrer. Uma abordagem primeiro-em-largura para esse tipo de teste não simularia *submetido* antes de simular *indefinido* e *definido*. Se o fizesse, faria uso das transições que não foram testadas previamente e, portanto, violaria o critério primeiro-em-largura.

19.7 RESUMO

O objetivo geral do teste orientado a objeto — encontrar o número máximo de erros com um mínimo de esforço — é idêntico ao objetivo do teste de software convencional. Mas as estratégias e as táticas para o teste orientado a objeto diferem significativamente. A visão do teste se amplia para incluir a revisão tanto dos requisitos quanto do modelo de projeto. Além disso, o foco do teste afasta-se do componente procedural (o módulo) e se aproxima da classe.

Em virtude dos requisitos e modelos de projeto orientados a objeto e o código fonte resultante serem semanticamente acoplados, o teste (na forma de revisões técnicas) começa durante a atividade de modelagem. Por essa razão, a revisão do CRC, da relação de objeto e dos modelos de comportamento pode ser vista como um teste de primeiro estágio.

Uma vez disponível o código, é aplicado o teste de unidade para cada classe. O projeto de testes para uma classe usa uma variedade de métodos: teste baseado em falha, teste aleatório e teste de partição. Cada um desses métodos simula as operações encapsuladas pela classe. Sequências de teste são projetadas para garantir que sejam exercitadas as operações relevantes. O estado da classe, representado pelos valores de seus atributos, é examinado para determinar se existem erros.

Teste de integração pode ser feito usando uma estratégia baseada em sequência de execução (*thread*) ou baseado em uso. O teste baseado em sequência de execução integra o conjunto de classes que colaboram para responder a uma entrada ou evento. O teste baseado em uso constrói o sistema em camadas, começando com aquelas classes que não fazem uso das classes servidoras. Métodos de projeto de caso de teste de integração podem também fazer uso de testes aleatórios e de partição. Além disso, testes baseados em cenário e derivados de modelos comportamentais podem ser usados para testar uma classe e seus colaboradores. Uma sequência de teste acompanha o fluxo de operações por meio das colaborações de classe.

Teste de validação orientado a objeto é teste orientado à caixa-preta e pode ser realizado aplicando-se os mesmos métodos caixa-preta discutidos para software convencional. No entanto, o teste baseado em cenário domina a validação de sistemas orientados a objeto, tornando o caso de uso um pseudocontrolador primário para teste de validação.

PROBLEMAS E PONTOS A PONDERAR

- 19.1. Com suas palavras, descreva por que a classe é a menor unidade razoável para teste em um sistema orientado a objeto.
- 19.2. Por que temos de retestar subclasses instanciadas de uma classe existente, se a classe existente já foi completamente testada? Podemos usar o projeto de caso de teste para a classe existente?
- 19.3. Por que o “teste” deveria começar com análise e projeto orientado a objeto?
- 19.4. Crie uma série de cartões de índice CRC para o *CasaSegura* e execute os passos descritos na Seção 19.2.2 para determinar se existem inconsistências.
- 19.5. Qual a diferença entre estratégias baseadas em sequências de execução (*thread*) e estratégias baseadas em uso para teste de integração? Como o teste de conjunto se encaixa?
- 19.6. Aplique teste aleatório e teste de particionamento a três classes definidas no projeto para o sistema *CasaSegura*. Produza casos de teste que indiquem as sequências de operação que serão chamadas.
- 19.7. Aplique teste de múltiplas classes e testes derivados de modelo comportamental para o projeto *CasaSegura*.
- 19.8. Crie quatro testes adicionais usando teste aleatório e métodos de particionamento, bem como teste de múltiplas classes e testes derivados do modelo comportamental para a aplicação bancária apresentada nas Seções 19.5 e 19.6.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Muitos livros sobre testes citados nas seções Leituras e Fontes de Informação Complementares dos Capítulos 17 e 18 discutem o teste de sistemas orientados a objeto até certo ponto. Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 6th ed., 2004) considera o teste orientado a objeto dentro do contexto mais amplo da prática da engenharia de software. Sykes and McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir and Goel (*Testing Object-Oriented Software*, Springer 2000), Binder (*Testing Object-Oriented Systems*, Addison-Wesley, 1999) e Kung e seus colegas (*Testing Object-Oriented Software*, Wiley-IEEE Computer Society Press, 1998) tratam o teste orientado a objeto em detalhes significativos.

Uma ampla gama de fontes de informação sobre métodos de teste orientado a objeto está disponível na Internet. Uma lista atualizada de referências na Web, relevante para as técnicas de teste, pode ser encontrada no site www.mhhe.com/engcs/compSci/pressman/professional/olc/ser.htm.

20

TESTANDO APLICAÇÕES PARA WEB

CONCEITOS-CHAVE

dimensões da qualidade	469
estratégias	470
planejamento	470
teste de base de dados	473
teste de carga	486
teste de compatibilidade	478
teste de configuração	482
teste de conteúdo	472
teste de desempenho	485
teste de esforço (stress)	486
teste de interface	474
teste de navegação	480
teste de segurança	484
testes de usabilidade	477
teste em nível de componente	479

Há uma urgência que sempre permeia um projeto de WebApp (aplicações para Web). Os grupos de interessados – preocupados com a competição de outras WebApps, coagidos pelas exigências do cliente e temerosos com a perda de um nicho de mercado — exercem pressão para obtê-la on-line. Como resultado, as atividades técnicas que muitas vezes ocorrem mais tarde no processo, tais como teste de WebApp, dispõem algumas vezes de um prazo muito curto. Isso pode ser um erro catastrófico. Para tanto, os membros da equipe precisam assegurar que cada artefato tenha alta qualidade. Wallace e seus colegas [Wal03] observam isso quando afirmam:

O teste não deveria esperar até que o projeto esteja terminado. Comece o teste antes de escrever uma linha de código. Teste constante e efetivamente, e você desenvolverá um site muito mais duradouro.

Como os requisitos e modelos de projeto não podem ser testados no sentido clássico, toda a equipe deve realizar revisões técnicas (Capítulo 15) e também testes executáveis. A intenção é descobrir e corrigir erros antes que a WebApp seja disponibilizada aos usuários finais.

PANORAMA

O que é? O teste de WebApp é um conjunto de atividades relacionadas com um único objetivo: descobrir erros no conteúdo, na função, na usabilidade, na navegabilidade, no desempenho, na capacidade e na segurança da WebApp. Para tanto, deve ser utilizada uma estratégia de teste que abrange as revisões e o teste executável.

Quem realiza? Os engenheiros de aplicações para Web e outros interessados no projeto (gerentes, clientes e usuários finais), todos participam do teste da WebApp.

Porque é importante? Se os usuários finais encontrarem erros que abalem sua credibilidade na WebApp, buscarão em outro lugar pelo conteúdo e função que precisam, e a WebApp será um fracasso. Por essa razão, deve-se trabalhar para eliminar tantos erros quantos forem possíveis antes que a WebApp entre no ar.

Quais são as etapas envolvidas? O processo de teste começa focalizando os aspectos visíveis da WebApp ao usuário e passa para os testes de tecnologia e infraestrutura. Executam-se sete etapas de teste: de conteúdo, interface, navegação, componente, configuração, desempenho e segurança.

Qual é o artefato? Em algumas instâncias um plano de teste de uma WebApp é produzido. Em qualquer instância, é desenvolvida uma série de casos de testes para todas as etapas e é mantido um arquivo dos resultados para uso futuro.

Como garantir que o trabalho foi realizado corretamente? Embora nunca se possa ter certeza de ter executado todos os testes necessários, é possível verificar se erros foram apontados (e esses foram corrigidos). Além disso, se foi estabelecido um plano de teste, é aconselhável certificar-se de que todos os testes planejados foram realizados.

20.1 CONCEITOS DE TESTE PARA WEBAPPS

O teste é um processo pelo qual se experimenta o software com a intenção de encontrar (e por fim corrigir) erros. Essa filosofia fundamental, apresentada inicialmente no Capítulo 17, não muda para as WebApps. Na verdade, pelo fato de os sistemas e aplicações baseados na Web residirem em uma rede e interoperarem com muitos sistemas operacionais diferentes, browsers (residindo em uma variedade de dispositivos), plataformas de hardware, protocolos de comunicação e aplicações “de retaguarda” à procura dos erros representam um desafio significativo para os engenheiros.

Para entender os objetivos do teste no contexto de engenharia de Web, deve-se considerar as muitas dimensões da qualidade da WebApp.¹ Aqui, consideramos dimensões de qualidade particularmente relevantes em qualquer discussão de teste da WebApp. Consideramos também a natureza dos erros encontrados como consequência do teste e a estratégia de teste aplicada para descobrir esses erros.

20.1.1 Dimensões da qualidade

A qualidade é incorporada a uma aplicação Web como consequência de um bom projeto. Ela é avaliada aplicando-se uma série de revisões técnicas que investigam vários elementos do modelo de projeto e utilizando-se um processo de teste que é discutido no decorrer deste capítulo. Revisões e teste examinam ambos uma ou mais das seguintes dimensões da qualidade [Mil00a]:

- *Conteúdo*: é avaliado em nível sintático e semântico. No nível sintático, examina-se a ortografia, pontuação e gramática em documentos baseados em texto. No nível semântico, são analisadas: exatidão (das informações apresentadas), consistência (através de todo o objeto de conteúdo e objetos relacionados) e ausência de ambiguidade.
- *Função*: é testada para descobrir erros que indicam falta de conformidade com os requisitos do cliente. Cada função da WebApp é avaliada quanto à exatidão, instabilidade e conformidade geral com os padrões apropriados de implementação (por exemplo, padrões de linguagem Java ou AJAX).
- *Estrutura*: é avaliada para assegurar o fornecimento apropriado de conteúdo e função da WebApp, que seja extensível e que possa ser mantido na medida em que novo conteúdo ou nova funcionalidade são acrescentados.
- *Usabilidade*: é testada para garantir que cada categoria de usuário seja suportada pela interface e que possa aprender e aplicar todas as sintaxes e semânticas de navegação necessárias.
- *Navegabilidade*: é testada para assegurar que toda a sintaxe e semânticas de navegação sejam experimentadas para descobrir quaisquer erros de navegação (por exemplo, links inativos, impróprios, errados).
- *Desempenho*: é testado sob uma variedade de condições de operação, configurações e carga para assegurar que o sistema responda à interação com o usuário e suporte cargas extremas sem degradação inaceitável da operação.
- *Compatibilidade*: é testada executando-se a WebApp em uma variedade de diferentes configurações hospedeiras tanto no lado cliente quanto no lado servidor. A finalidade é encontrar erros específicos a determinada configuração de hospedeira.
- *Interoperabilidade*: é testada para garantir que a WebApp tenha uma interface adequada com outras aplicações e/ou bases de dados.
- *Segurança*: é testada para investigar vulnerabilidades potenciais e tentar explorar cada uma delas. Qualquer tentativa de invasão bem-sucedida é considerada uma falha de segurança.

Desenvolveram-se estratégias e táticas para teste de WebApp para experimentar cada uma dessas dimensões da qualidade. Posteriormente, neste capítulo, elas serão discutidas.

20.1.2 Erros em um ambiente WebApp

Os erros encontrados em consequência de um teste de WebApp bem-sucedido têm uma série de características especiais [Ngu00]:

 **Como investigamos a qualidade no contexto de uma WebApp e seu ambiente?**

“Inovação é o paraíso para os testadores de software. Justamente quando parece que sabemos como testar uma tecnologia em particular, uma nova tecnologia [WebApps] aparece e todas as possibilidades perdem sentido.”

James Bach

¹ As dimensões de qualidade de software genérico, igualmente aplicável a WebApps, são discutidas no Capítulo 14.

? O que torna os erros encontrados durante a execução de WebApp de certa forma diferentes daqueles identificados em software convencional?

1. Devido a muitos tipos de testes de WebApp detectarem problemas evidenciados primeiro no lado cliente (isto é, via interface implementada em um browser específico ou em um dispositivo de comunicação pessoal), muitas vezes nos deparamos com o indício de erro e não o erro em si.
2. Devido a uma WebApp ser implementada em diferentes configurações e ambientes, pode ser difícil ou impossível reproduzir um erro fora do ambiente no qual foi encontrado originalmente.
3. Embora alguns erros sejam resultado de projeto incorreto ou codificação HTML (ou outra linguagem de programação) incorreta, muitos erros podem ser atribuídos à configuração da WebApp.
4. Devido a WebApps residirem em uma arquitetura cliente-servidor, os erros podem ser difíceis de localizar através das três camadas arquitetônicas: o cliente, o servidor ou a própria rede.
5. Alguns erros são decorrentes do *ambiente operacional estático* (a configuração específica na qual o teste é executado), enquanto outros são atribuíveis ao ambiente operacional dinâmico (a carga instantânea de recursos ou erros relacionados com o tempo).

Esses cinco atributos de erros sugerem que o ambiente tem um papel importante no diagnóstico de todos os erros encontrados durante o teste da WebApp. Em algumas situações (por exemplo, teste de conteúdo), o local dos erros é óbvio, mas em muitos outros tipos de teste da WebApp (por exemplo, teste de navegação, teste de desempenho, teste de segurança) a causa subjacente do erro pode ser consideravelmente mais difícil de determinar.

20.1.3 Estratégia de teste

A estratégia para teste de WebApp adota os princípios básicos para todo o teste de software (Capítulo 17) e aplica estratégia e táticas recomendadas para sistemas orientados a objeto (Capítulo 19). Os passos a seguir resumem a abordagem:

1. O modelo de conteúdo para a WebApp é revisto para descobrir erros.
2. O modelo de interface é revisto para garantir que todos os casos de uso possam ser acomodados.
3. O modelo de projeto da WebApp é revisto para descobrir erros de navegação.
4. A interface com o usuário é testada para descobrir erros nos mecanismos de apresentação e/ou navegação.
5. Os componentes funcionais são submetidos a testes de unidade.
6. É testada a navegação por toda a arquitetura.
7. A WebApp é implementada em uma variedade de configurações ambientais diferentes e testada quanto à compatibilidade com cada configuração.
8. São executados testes de segurança na tentativa de explorar vulnerabilidades na WebApp ou em seu ambiente.
9. São realizados testes de desempenho.
10. A WebApp é testada por uma população de usuários finais controlados e monitorados; os resultados de suas interações com o sistema são avaliados quanto a erros de conteúdo e navegação, preocupações de usabilidade, preocupações de compatibilidade, segurança, confiabilidade e desempenho da WebApp.

Devido ao fato de muitas WebApps evoluírem continuamente, o processo de teste é uma atividade contínua, executada por pessoal de suporte que usa testes de regressão derivados dos desenvolvidos quando a WebApp foi inicialmente criada.

20.1.4 Planejamento de teste

O uso da palavra *planejamento* (em qualquer contexto) é anátema para alguns desenvolvedores da Web. Estes não planejam; simplesmente iniciam — na esperança de que irá surgir uma poderosa WebApp. Uma abordagem mais disciplinada reconhece que o planejamento estabelece um roteiro para todo o trabalho a ser feito. Isso compensa o esforço. Em seu livro sobre teste de WebApp, Splaine e Jaskiel [Spl01] afirmam:

PONTO-CHAVE

A estratégia geral para teste de WebApp pode ser resumida nos dez passos descritos a seguir.

WebRef

Excelentes artigos sobre teste de WebApp podem ser encontrados em www.stickyminds.com/testing.asp.

PONTO-CHAVE

O plano de teste identifica um conjunto de tarefas de teste, os artefatos a ser desenvolvidos e a maneira pela qual os resultados devem ser avaliados, registrados e reutilizados.

Exceto para o mais simples dos sites, torna-se imediatamente aparente que algum tipo de planejamento de teste seja necessário. Com muita frequência, o número inicial de erros encontrados por meio de um teste *ad hoc* é tão grande que nem todos são corrigidos quando detectados. Isso apresenta uma dificuldade adicional para os testadores de sites e aplicações. Eles devem não apenas evocar novos testes criativos, mas também lembrar como os anteriores foram executados para retestar o site/aplicação de forma confiável e assegurar que os erros conhecidos tenham sido removidos e que novos erros não tenham sido introduzidos.

As perguntas a ser feitas são: como “podemos evocar novos testes criativos”, e o que esses testes devem focalizar? As respostas a essas perguntas estão contidas em um plano de teste.

Um plano de teste da WebApp identifica (1) o conjunto de tarefas² a ser aplicado quando o teste começa, (2) os artefatos que serão produzidos na medida em que cada tarefa de teste é executada e (3) a maneira pela qual os resultados do teste são avaliados, registrados e reutilizados quando for executado o teste de regressão. Em alguns casos, o plano de teste é integrado ao plano do projeto. Em outros, o plano de teste é um documento separado.

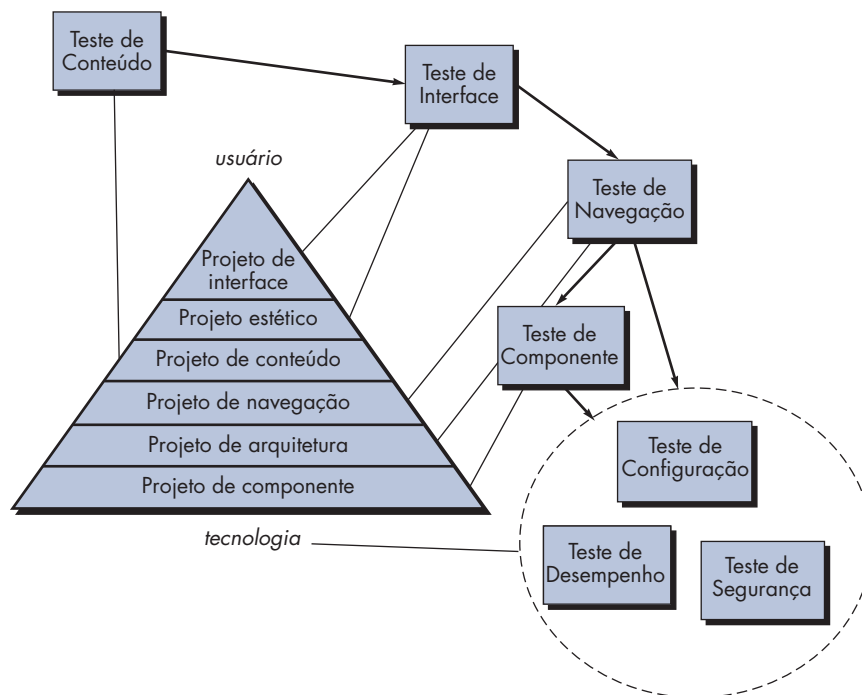
20.2 O PROCESSO DE TESTE – UMA VISÃO GERAL

O processo de teste da WebApp começa com testes que experimentam o conteúdo e a funcionalidade da interface, pontos imediatamente visíveis aos usuários finais. Na medida em que o teste é realizado, são experimentados aspectos da arquitetura de projeto e da navegação da WebApp. Por fim, o foco muda para testes que examinam os recursos tecnológicos que nem sempre são aparentes aos usuários finais – tópicos de infraestrutura e de instalação/implementação da WebApp.

A Figura 20.1 aproxima o processo de teste da WebApp da pirâmide de projeto para WebApps (Capítulo 13). Observe que enquanto ocorre o fluxo do teste da esquerda para a direita e de cima para baixo, os elementos do projeto da WebApp visíveis ao usuário (elementos do topo da pirâmide) são testados primeiro, seguidos pelos elementos de projeto da infraestrutura.

FIGURA 20.1

O processo de teste



² Conjuntos de tarefas são discutidos no Capítulo 2. Um termo relacionado – *fluxo de trabalho* – é também usado para descrever uma série de tarefas necessárias para exercer uma atividade de engenharia de software.

20.3 TESTE DE CONTEÚDO



Embora as revisões técnicas não façam parte do teste, a de conteúdo deverá ser executada para garantir que o conteúdo tenha qualidade.

Erros no conteúdo da WebApp podem ser tão triviais quanto pequenos erros tipográficos ou muito significativos como informações incorretas, organização inadequada ou violação de leis de propriedade intelectual. O teste de conteúdo tenta descobrir esses e muitos outros problemas antes que sejam encontrados pelos usuários.

Ele combina tanto revisões quanto geração de casos de testes executáveis. As revisões são aplicadas para descobrir erros semânticos em conteúdo (discutido na Seção 20.3.1). O teste executável é usado para descobrir erros de conteúdo que podem ser atribuídos a conteúdo derivado dinamicamente, controlado por dados adquiridos de um ou mais banco de dados.

20.3.1 Objetivos do teste de conteúdo

O teste de conteúdo tem três importantes objetivos: (1) descobrir erros de sintaxe (por exemplo, erros ortográficos, erros gramaticais) em documentos de texto, representações gráficas e outros meios; (2) descobrir erros de semântica (isto é, erros na exatidão ou integralidade das informações) em qualquer objeto de conteúdo apresentado quando ocorre a navegação e (3) encontrar erros na organização ou estrutura do conteúdo apresentado ao usuário final.

Para atingir o primeiro objetivo, usam-se verificadores automáticos de ortografia e gramática. Porém, muitos erros de sintaxe fogem à detecção por essas ferramentas e devem ser descobertos por um revisor humano (testador). De fato, um grande site pode contratar os serviços de um revisor profissional para descobrir erros de ortografia, erros de gramática, erros na consistência do conteúdo, erros em representações gráficas e erros de referência cruzada.

O teste de semântica concentra-se nas informações apresentadas em cada objeto de conteúdo. O revisor (testador) deve responder às seguintes questões:

- As informações são efetivamente precisas?
- As informações são concisas e direcionadas ao assunto?
- É fácil para o usuário entender o layout do objeto de conteúdo?
- As informações contidas em um objeto de conteúdo podem ser encontradas facilmente?
- Foram fornecidas referências apropriadas para todas as informações derivadas de outras fontes?
- As informações apresentadas são consistentes internamente e consistentes com as informações apresentadas em outros objetos de conteúdo?
- O conteúdo é ofensivo, confuso ou dá margem a litígio?
- O conteúdo desrespeita os direitos autorais existentes ou de marcas registradas?
- O conteúdo contém links que complementam o conteúdo existente? Os links estão corretos?
- O estilo estético do conteúdo está em conflito com o estilo estético da interface?

Obter respostas para cada uma dessas perguntas para uma grande WebApp (contendo centenas de objetos de conteúdo) pode ser uma tarefa assustadora. No entanto, se não forem descobertos os erros de semântica, será abalada a confiança do usuário na WebApp e isso pode levar ao fracasso da aplicação.

Objetos de conteúdo existem em uma arquitetura que tem um estilo específico (Capítulo 13). Durante o teste de conteúdo, a estrutura e a organização da arquitetura de conteúdo são verificadas para assegurar que o conteúdo necessário seja apresentado ao usuário final na ordem e relacionamentos apropriados. Por exemplo, a WebApp **CasaSeguraGarantida.com** apresenta uma variedade de informações sobre sensores usados como parte dos produtos de segurança e vigilância. Objetos de conteúdo fornecem informações descritivas, especificações técnicas, uma representação fotográfica e informações relacionadas. Os testes da arquitetura de conteúdo de **CasaSeguraGarantida.com** procuram descobrir erros na apresentação dessas informações (por exemplo, uma descrição do Sensor X é apresentada com uma foto do Sensor Y).

PONTO-CHAVE

Os objetivos do teste de conteúdo são: (1) descobrir erros de sintaxe no conteúdo, (2) descobrir erros de semântica e (3) encontrar erros estruturais.



Quais questões deverão ser formuladas e respondidas para descobrir erros de semântica no conteúdo?

“Em geral, as técnicas de teste de software usadas em outras aplicações são as mesmas que as utilizadas em aplicações baseadas na Web... A diferença é que a tecnologia varia na multiplicidade do ambiente Web.”

Hung Nguyen

20.3.2 Teste de base de dados

As WebApps modernas fazem muito mais do que apresentar objetos de conteúdo estáticos. Em muitos domínios de aplicação, interfaceiam com sistemas sofisticados de gerenciamento de banco de dados e criam objetos de conteúdo dinâmico em tempo real usando os dados adquiridos de um banco de dados.

Por exemplo, uma WebApp de serviços financeiros pode produzir informações complexas baseadas em texto, na forma tabular e gráfica sobre um fundo específico (por exemplo, um fundo de ações ou fundo mútuo). O objeto de conteúdo composto que apresenta essas informações é criado dinamicamente quando o usuário solicita informações sobre um fundo específico. Para tanto, são necessários os seguintes passos: (1) é consultado um grande banco de dados de fundos, (2) são extraídos os dados relevantes do banco de dados, (3) os dados extraídos devem ser organizados como um objeto de conteúdo e (4) esse objeto de conteúdo (representando informações personalizadas requisitadas por um usuário final) é transmitido para o ambiente do cliente para ser apresentado. Erros podem ocorrer e efetivamente ocorrem, em consequência de cada uma dessas etapas. O objetivo do teste de banco de dados é descobrir os erros, mas esse tipo de teste é complicado por uma variedade de fatores:

 **Quais os tópicos que complicam o teste de banco de dados para WebApps?**

1. *A solicitação original de informações do lado do cliente raramente é apresentada de maneira (por exemplo, linguagem de consulta estruturada, Structured Query Language – SQL) que possa ser colocada em um sistema de gerenciamento de banco de dados (database management system – DBMS). Portanto, deverão ser projetados testes para descobrir erros de tradução da solicitação do usuário em uma forma que possa ser processada pelo DBMS.*
2. *O banco de dados pode ser remoto ao servidor que abriga a WebApp. Portanto, devem ser desenvolvidos testes que descubram erros na comunicação entre a WebApp e o banco de dados remoto.*³
3. *Dados brutos adquiridos do banco de dados devem ser transmitidos para o servidor da WebApp e formatados corretamente para subsequente transmissão ao cliente. Portanto, devem ser criados testes que demonstrem a validade dos dados brutos recebidos pelo servidor da WebApp, e devem ser criados também testes adicionais que demonstram a validade das transformações aplicadas a esses dados para criar objetos de conteúdo válidos.*
4. *O(s) objeto(s) de conteúdo dinâmico deve(m) ser transmitido(s) ao cliente de maneira que possa(m) ser apresentado(s) ao usuário final. Portanto, uma série de testes deverá ser projetada para (1) descobrir erros no formato do objeto de conteúdo e (2) testar compatibilidade com diferentes configurações de ambiente.*

“... nós não confiamos em um site que sofre de frequentes paradas, trava no meio de uma transação ou tem pouco senso de utilidade. O teste, portanto, tem um papel crucial no processo geral de desenvolvimento.”

Wing Lam

Considerando esses quatro fatores, os métodos de projeto de casos de teste deverão ser aplicados a cada uma das “camadas de interação” [Ngu01] observadas na Figura 20.2. O teste deve assegurar que (1) sejam passadas informações válidas entre o cliente e o servidor por meio da camada de interface, (2) que a WebApp processe scripts corretamente e extraia ou formata adequadamente os dados do usuário, (3) que os dados do usuário são passados corretamente para uma função de transformação de dados no lado servidor que formata apropriadamente as consultas (por exemplo, SQL), (4) que as consultas são passadas para uma camada de gerenciamento de dados⁴ que se comunica com as rotinas de acesso a banco de dados (potencialmente localizadas em outras máquinas).

As camadas de transformação de dados, gerenciamento de dados e acesso a banco de dados da Figura 20.2 muitas vezes são construídas com componentes reutilizáveis que foram validados separadamente e como um pacote. Se for esse o caso, o teste da WebApp focaliza o projeto de casos de testes para experimentar as interações entre a camada cliente e as duas primeiras camadas servidor (WebApp e transformação de dados) mostradas na figura.

³ Esses testes podem se tornar complexos quando são encontradas bases de dados distribuídas ou quando é necessário acesso a um armazém de dados (Capítulo 1).

⁴ A camada de gerenciamento de dados tipicamente incorpora uma interface SQL em nível de chamada (SQL-CLI) como, por exemplo, o Microsoft OLE/ADO ou Java Database Connectivity (JDBC).

A camada da interface de usuário é testada para garantir que os scripts sejam construídos corretamente para cada consulta de usuário e adequadamente transmitidos para o lado servidor. A camada da WebApp no lado do servidor é testada para assegurar que os dados do usuário sejam adequadamente extraídos dos scripts e transmitidos corretamente para a camada de transformação de dados no lado do servidor. As funções de transformação de dados são testadas para assegurar que o SQL correto seja criado e passado para os componentes apropriados de gerenciamento de dados.

Uma discussão detalhada da tecnologia subjacente que deve ser entendida para projetar adequadamente esses testes de banco de dados está além do escopo deste livro. Para mais detalhes, veja [Sce02], [Ngu01] e [Bro01].

20.4 TESTE DA INTERFACE DE USUÁRIO



Com exceção de aspectos específicos orientados para a WebApp, a estratégia de interface observada aqui é aplicável a todos os tipos de software cliente-servidor.

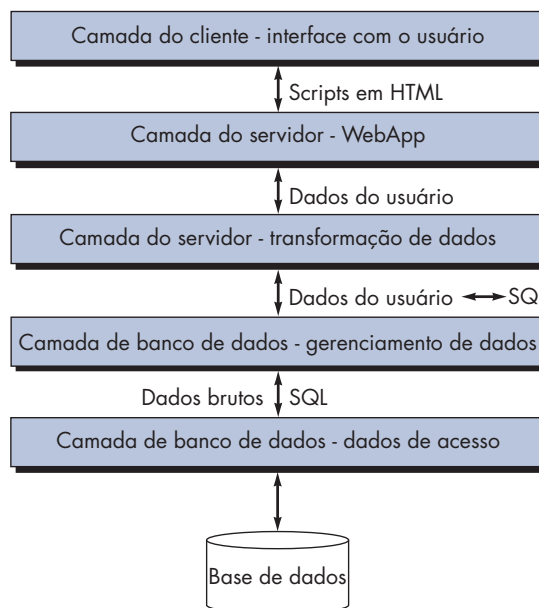
A verificação e a validação de uma interface de usuário de uma WebApp ocorre em três pontos distintos. Durante a análise, o modelo de interface é revisado para assegurar que esteja em conformidade com os requisitos dos interessados e com outros elementos do modelo de requisitos. Durante o projeto, o modelo de projeto de interface é revisado para assegurar que critérios genéricos de qualidade estabelecidos para todas as interfaces de usuário (Capítulo 11) tenham sido satisfeitos e que os problemas de projeto de interface específicos da aplicação tenham sido corretamente resolvidos. Durante o teste, o foco passa para a execução de aspectos da interação com o usuário específico da aplicação à medida que são manifestados pela sintaxe e semântica da interface. Além disso, o teste fornece uma avaliação final de usabilidade.

20.4.1 Estratégia de teste de interface

O teste de interface experimenta mecanismos de interação e valida aspectos estéticos da interface de usuário. A estratégia geral é (1) descobrir erros relacionados com mecanismos específicos de interface (por exemplo, erros na execução apropriada de um link de menu ou na maneira como os dados são colocados em um formulário) e (2) descobrir erros na maneira como a interface implementa as semânticas de navegação, funcionalidade da WebApp ou exibição de conteúdo. Para essa estratégia, há uma série de objetivos a ser atingidos:

FIGURA 20.2

Camadas de interação



- *Características de interface são testadas para garantir que as regras de projeto, estética e conteúdo visual relacionado estejam disponíveis para o usuário sem erro.* As características incluem fontes, o uso da cor, molduras, imagens, bordas, tabelas e características de interface relacionadas que são geradas quando ocorre a execução da WebApp.
- *Mecanismos individuais de interface são testados de uma maneira análoga ao teste de unidade.* Por exemplo, são projetados testes para experimentar todos os formulários, scripts no lado do cliente, HTML dinâmico, scripts, conteúdo concatenado e mecanismos de interface específicos da aplicação (por exemplo, um carrinho de compras para uma aplicação de e-commerce). Em muitos casos, o teste pode concentrar-se exclusivamente em um desses mecanismos (a “unidade”) excluindo todas as outras características e funções de interface.
- *Cada mecanismo de interface é testado de acordo com o contexto de um caso de uso ou uma unidade semântica de navegação, NSU, (Capítulo 13) para uma categoria específica de usuário.* Essa abordagem é análoga ao teste de integração porque os testes são executados à medida que os mecanismos de interface são integrados para permitir que um caso de uso ou uma NSU sejam executados.
- *A interface completa é testada em relação aos casos de uso selecionados e NSUs para descobrir erros nas semânticas da interface.* Essa abordagem é análoga ao teste de validação porque a finalidade é demonstrar conformidade com semânticas específicas de casos de uso ou de NSU. É nesse estágio que uma série de testes de usabilidade é executada.
- *A interface é testada segundo uma variedade de ambientes (por exemplo, navegadores) para assegurar que sejam compatíveis.* Na realidade, essas séries de testes podem ser consideradas como parte do teste de configuração.



O teste de links externos deve ocorrer durante toda a vida da WebApp. Como parte de estratégia de manutenção, testes de links devem ser agendados regularmente.

20.4.2 Testando mecanismos de interface

Quando um usuário interage com uma WebApp, a interação ocorre através de um ou mais mecanismos de interface. Uma rápida revisão das considerações para cada mecanismo de interface é apresentada nos próximos parágrafos [Spl01].

Links. Cada link de navegação é testado para assegurar que o objeto de conteúdo ou a função apropriados sejam acessados⁵. Faz-se uma lista de todos os links associados com o layout da interface (por exemplo, barras de menu, itens de índice) e então se executa cada um individualmente. Além disso, os links em cada objeto de conteúdo devem ser experimentados para descobrir as URLs incorretas ou links para objetos de conteúdo ou funções impróprios. Por fim, links para WebApps externas deverão ser testados quanto à exatidão e avaliados para determinar o risco de se tornarem inválidos com o tempo.

Formulários. Em nível macroscópico, são executados testes para assegurar que (1) rótulos identifiquem corretamente os campos no formulário e que os campos obrigatórios sejam identificados visualmente para o usuário, (2) o servidor receba todas as informações contidas no formulário e que não seja perdido nenhum dado na transmissão entre o cliente e o servidor, (3) sejam usadas as escolhas-padrão (*defaults*) apropriadas quando o usuário não seleciona de um menu pull down ou por meio de uma série de botões, (4) funções do navegador (por exemplo, a seta de retorno) não corrompam os dados introduzidos em um formulário e (5) scripts que realizam verificação de erros sobre dados introduzidos funcionem adequadamente e proporcionem mensagens de erros coerentes.

Em um nível mais específico, os testes devem assegurar que (1) os campos do formulário tenham tamanho e tipos de dados corretos, (2) o formulário estabeleça proteções apropriadas que impedem que o usuário digite cadeias de texto mais longas do que um comprimento máximo predefinido, (3) todas as opções apropriadas para menus pull-down sejam especificadas e ordenadas de maneira que tenha significado para o usuário final, (4) os recursos de “preenchimento

⁵ Esses testes podem ser realizados tanto como parte do teste de interface quanto do de navegação.



Os testes de script no lado do cliente e testes associados com HTML dinâmico deverão ser repetidos sempre que uma nova versão de navegador popular é lançada.

automático” do navegador não causem erros nas entradas de dados e (5) a tecla tab (ou alguma outra tecla) faça a movimentação apropriada entre os campos do formulário.

Script no lado do cliente. Os testes caixa-preta são realizados para descobrir quaisquer erros no processamento quando o script é executado. Eles muitas vezes são acoplados ao teste de formulário, porque a entrada de script é em geral derivada de dados fornecidos como parte de processamento de formulários. Deverá ser feito um teste de compatibilidade para assegurar que a linguagem de script escolhida funcionará corretamente nas configurações ambientais que suportam a WebApp. Além disso, para testar o próprio script, Splaine e Jaskiel [Spl01] sugerem que “você deve assegurar que os padrões de sua empresa [WebApp] definam a linguagem preferida e a versão de linguagem de script a ser usada para o lado do cliente (e o lado do servidor)”.

HTML dinâmico. Cada página Web que contenha HTML dinâmico é executada para assegurar que a exibição dinâmica esteja correta. Além disso, deverá ser feito um teste de compatibilidade para garantir que o HTML funcione corretamente nas configurações ambientais que suportam a WebApp.

Janelas pop-up. Uma série de testes garante que (1) o pop-up esteja dimensionado e posicionado corretamente, (2) o pop-up não cubra a janela original da WebApp, (3) o projeto estético do pop-up seja consistente com o da interface e (4) barras de rolagem e outros mecanismos de controle acrescentados ao pop-up estejam corretamente localizados e funcionem conforme desejado.

Scripts CGI. São feitos testes caixa-preta com ênfase na integridade dos dados (quando os dados são passados para o script CGI) e processamento de script (uma vez que dados validados tenham sido recebidos). Além disso, pode ser feito teste de desempenho para assegurar que a configuração do lado servidor possa acomodar as demandas de processamento de múltiplas chamadas de scripts CGI [Spl01].

Conteúdo encadeado (streaming). Os testes devem demonstrar que os dados encadeados são atualizados, exibidos corretamente e que podem ser suspensos sem erro e reiniciados sem dificuldade.

Cookies. São necessários testes do lado do servidor e do lado do cliente. No lado do servidor, deverão assegurar que um cookie esteja corretamente construído (contém dados corretos) e corretamente transmitido para o lado do cliente, quando solicitado um conteúdo ou uma funcionalidade específica. Além disso, a persistência apropriada do cookie é testada para assegurar que sua data de expiração esteja correta. Do lado do cliente, testes determinam se a WebApp anexa corretamente os cookies existentes de acordo com a solicitação específica (enviada ao servidor).

Mecanismos de interface específicos de aplicativo. Os testes seguem uma checklist de funcionalidade e características definidas pelo mecanismo de interface. Por exemplo, Splaine e Jaskiel [Spl01] sugerem a seguinte checklist para funcionalidade do carrinho de compras definido para uma aplicação de e-commerce:

- Faça o teste de fronteiras (Capítulo 18) do número mínimo e máximo de itens que podem ser colocados no carrinho de compras.
- Teste uma solicitação de “saída” para um carrinho de compras vazio.
- Teste a remoção apropriada de um item do carrinho de compras.
- Teste para determinar se uma compra esvazia todo o conteúdo de um carrinho de compras.
- Teste para determinar a persistência do conteúdo do carrinho de compras (isso deverá ser especificado como parte dos requisitos do cliente).
- Teste para determinar se a WebApp pode restaurar o conteúdo do carrinho de compras em alguma data futura (supondo que não foi feita nenhuma compra).

20.4.3 Testando semânticas de interface

Uma vez que cada mecanismo de interface foi testado em termos de “unidade”, o foco muda para as semânticas de interface. O teste de semânticas de interface “avalia quão bem o projeto se preocupa com os usuários, oferece diretrizes claras, fornece realimentação e mantém consistência de linguagem e abordagem” [Ngu00].

Uma revisão rigorosa do modelo de projeto de interface pode proporcionar respostas parciais às questões do parágrafo anterior. No entanto, cada cenário de caso de uso (para cada categoria de usuário) deverá ser testado uma vez implementada a WebApp. Essencialmente, um caso de uso torna-se a entrada para o projeto de uma sequência de testes. A finalidade da sequência de testes é descobrir erros que impedirão o usuário de atingir o objetivo associado ao caso de uso.

À medida que cada caso de uso é testado, é aconselhável manter uma checklist para garantir que todos os itens do menu foram usados pelo menos uma vez e que todos os links contidos em um objeto de conteúdo tenham sido utilizados. Além disso, a série de testes deve incluir seleção de opções de menu e uso de link inadequados. A finalidade é determinar se a WebApp proporciona manipulação eficaz do erro e recuperação.

20.4.4 Testes de usabilidade

WebRef

Um guia conveniente para o teste de usabilidade pode ser encontrado em www.ahref.com/guides/design/199806/0615jef.html.

Os testes de usabilidade são similares aos de semânticas de interface (Seção 20.4.3), porque também avaliam o grau com o qual os usuários podem interagir efetivamente com a WebApp e o grau em que a WebApp dirige as ações do usuário, proporciona uma boa realimentação e reforça uma abordagem de interação consistente. Em vez de concentrar-se intencionalmente nas semânticas de algum objetivo interativo, as revisões e testes de usabilidade são projetadas para determinar o grau com o qual a interface da WebApp facilita a vida do usuário.⁶

Invariavelmente o projetista contribuirá para o projeto dos testes de usabilidade, mas eles serão feitos pelos usuários finais. É aplicada a seguinte sequência de passos [Spl01]:

1. Definir uma série de categorias de teste de usabilidade e identificar objetivos para cada uma delas.
2. Projetar testes que permitirão avaliar cada um dos objetivos.
3. Selecionar os participantes que farão os testes.
4. Instrumentar a interação dos participantes com a WebApp enquanto o teste está em execução.
5. Desenvolver um mecanismo para avaliar a usabilidade da WebApp.

O teste de usabilidade pode ocorrer em diferentes níveis de abstração: (1) pode ser investigada a usabilidade de um mecanismo específico de interface (por exemplo, um formulário), (2) pode ser investigada a usabilidade de uma página Web completa (abrangendo mecanismos de interface, objetos de dados e funções relacionadas) ou (3) pode ser considerada a usabilidade da WebApp completa.

O primeiro passo no teste de utilidade é identificar uma série de categorias de usabilidade e estabelecer objetivos para cada uma das categorias. As seguintes categorias de teste e objetivos (escritos na forma de perguntas) ilustram tal abordagem:⁷

Interatividade — Os mecanismos de interação (por exemplo, menus desdobráveis, botões, ponteiros) são fáceis de entender e usar?

Layout — Os mecanismos de navegação, conteúdo e funções são colocados de maneira que permita ao usuário encontrá-los rapidamente?

? Quais características da usabilidade tornam-se o foco do teste e quais objetivos específicos são considerados?

6 O termo amigável ao usuário tem sido usado nesse contexto. O problema, naturalmente, é que a percepção de um usuário sobre uma interface “amigável” pode ser radicalmente diferente da de outro usuário.

7 Para mais informações sobre usabilidade, veja o Capítulo 11.

Clareza — O texto é bem escrito e fácil de ser entendido?⁸ As representações gráficas são fáceis de entender?

Estética — O layout, a cor, o tipo de letra e características relacionadas facilitam o uso? Os usuários se sentem “confortáveis” com a aparência e comportamento da WebApp?

Características da tela — A WebApp otimiza o uso do tamanho da tela e da resolução?

Sensibilidade ao tempo — Características importantes, funções e conteúdo podem ser usados ou acessados no tempo correto?

Personalização — A WebApp se adapta a necessidades específicas de diferentes categorias de usuário ou de usuários individuais?

Acessibilidade — A WebApp é acessível a pessoas com necessidades especiais?

Em cada uma dessas categorias é projetada uma série de testes. Em alguns casos, o “teste” pode ser uma revisão visual de uma página Web. Em outros, podem novamente ser executados testes de semânticas de interface, mas nesse caso os problemas de usabilidade são essenciais.

Como exemplo, consideramos a avaliação da usabilidade para mecanismos de interação e interface. Constantine e Lockwood [Con99] sugerem que a seguinte lista de características de interface deveria ser revista e testada quanto à usabilidade: animação, botões, cores, controle, diálogo, campos, formulários, molduras, gráficos, rótulos, links, menus, mensagens, páginas de navegação, seletores, texto e barras de ferramentas. À medida que cada característica é avaliada, ela é classificada em uma escala qualitativa pelos usuários que estão fazendo o teste. A Figura 20.3 mostra um conjunto possível de “graus” de avaliação que podem ser selecionados pelos usuários. Esses graus são aplicados a cada característica individualmente, para uma página Web completa, ou para a WebApp como um todo.

PONTO-CHAVE

As WebApps executam em uma variedade de ambientes do lado do cliente. O objetivo dos testes de compatibilidade é descobrir erros associados a um ambiente específico (por exemplo, navegador).

20.4.5 Testes de compatibilidade

Diferentes computadores, dispositivos de imagem, sistemas operacionais, navegadores e velocidades de conexão de rede podem ter influência significativa na operação da WebApp. Cada configuração de computador pode resultar em diferenças nas velocidades de processamento no lado do cliente, resoluções de tela e velocidades de conexão. Excentricidades de sistemas operacionais podem causar problemas no processamento da WebApp. Diferentes navegadores às vezes produzem resultados ligeiramente diferentes, independentemente do grau de padronização HTML na WebApp. Os plug-ins necessários podem ou não estar disponíveis para uma determinada configuração.

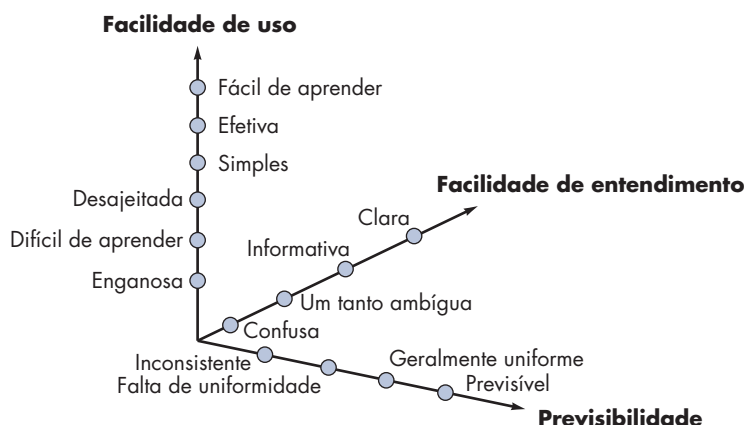
Em alguns casos, pequenos problemas de compatibilidade não apresentam dificuldades significativas, mas, em outros, podem ser encontrados erros graves. Por exemplo, velocidades de download podem se tornar inaceitáveis, a falta de um plug-in necessário pode tornar o conteúdo indisponível, diferenças entre navegadores podem mudar significativamente o layout da página, estilos de fontes podem ser alterados tornando o texto ilegível ou formulários podem ser organizados de forma inadequada. O teste de compatibilidade procura descobrir esses problemas antes que a WebApp entre no ar (fique on-line).

O primeiro passo no teste de compatibilidade é definir uma série de configurações de computadores “comumente encontradas” no lado do cliente e suas variantes. Essencialmente, é criada uma estrutura em árvore, identificando cada plataforma de computador, dispositivos típicos de imagem, sistemas operacionais suportados na plataforma, navegadores disponíveis, velocidades prováveis de conexão à internet e informações similares. Em seguida, é derivada uma série de testes de validação de compatibilidade, muitas vezes adaptados de testes de interfaces existentes, testes de navegação, testes de desempenho e testes de segurança. A finalidade desses testes é descobrir erros ou problemas de execução que podem ser atribuídos a diferenças em configuração.

⁸ O FOG Readability Index e outros pode ser usado para dar uma visão quantitativa da clareza. Veja mais detalhes em <http://developer.gnome.org/documents/usability/usability-reability.html>.

FIGURA 20.3

Avaliação qualitativa da usabilidade



20.5 TESTE NO NÍVEL DE COMPONENTE

O teste no nível de componente, também chamado de teste de função, concentra-se em um conjunto de testes que tentam descobrir erros nas funções da WebApp. Cada função da WebApp é um componente de software (implementado dentro de uma variedade de linguagens de programação ou scripts) e pode ser testado por meio de técnicas de caixa-preta (e em alguns casos, caixa-branca) discutidas no Capítulo 18.

CASASEGURA



Teste de WebApp

Cena: Escritório de Doug Miller.

Personagens: Doug Miller (gerente do grupo de engenharia de software CasaSegura) e Vinod Raman (membro da equipe de engenharia de software do produto).

Conversa:

Doug: O que você acha da WebApp de e-commerce CasaSeguraGarantida.com V.0.0?

Vinod: O prestador de serviço do fornecedor fez um bom trabalho. Sharon [gerente de desenvolvimento do fornecedor] disse-me que estão testando conforme conversamos.

Doug: Gostaria que você e os demais membros da equipe fizessem um pequeno teste informal no site de e-commerce.

Vinod (fazendo careta): Pensei que iríamos contratar uma empresa de teste de terceiros para validar a WebApp. Ainda estamos tentando liberar esse artefato.

Doug: Vamos contratar um fornecedor para testar o desempenho e segurança, e nosso fornecedor já está testando. Apenas pensei que outro ponto de vista seria útil e, além disso, precisamos manter os custos dentro dos limites, portanto...

Vinod (suspirando): O que você está procurando?

Doug: Quero ter certeza de que a interface e toda a navegação estão sólidas.

Vinod: Suponho que podemos começar com os casos de uso para cada uma das principais funções de interface:

Conheça o CasaSegura.

Especifique o sistema CasaSegura de que você precisa.

Compre um sistema CasaSegura.

Obtenha suporte técnico.

Doug: Bom. Mas percorra os caminhos de navegação até o fim.

Vinod (examinando um caderno de casos de uso): Sim, quando você seleciona **Especifique o sistema CasaSegura de que você precisa**, isso vai levá-lo a:

Selecione componentes do CasaSegura e Obtenha recomendações de componente do CasaSegura.

Podemos exercitar as semânticas de cada caminho.

Doug: Aproveite para verificar o conteúdo que aparece em cada nó de navegação.

Vinod: Claro... E os elementos funcionais também. Quem está testando a usabilidade?

Doug: Humm... O fornecedor de teste coordenará o teste de usabilidade. Nós contratamos uma empresa de pesquisa de mercado para selecionar 20 usuários típicos para o estudo de usabilidade, mas se vocês descobrirem quaisquer problemas de usabilidade...

Vinod: Eu sei, passamos para eles.

Doug: Obrigado, Vinod.

Casos de testes em nível de componente muitas vezes são controlados por entrada no nível de formulários. Uma vez definidos os dados dos formulários, o usuário seleciona um botão ou outro mecanismo de controle para iniciar a execução. Os seguintes métodos de projeto de casos de teste (Capítulo 18) são típicos:

- *Particionamento de equivalência* – O domínio de entrada da função é dividido em categorias de entrada ou classes a partir das quais são derivados os casos de testes.
- O formulário de entrada é analisado para determinar que classes de dados são relevantes para a função. Criam-se e executam-se casos de testes para cada classe de entrada, enquanto outras classes de entrada são mantidas constantes. Por exemplo, uma aplicação de e-commerce pode implementar uma função que calcula as despesas de envio. Entre uma variedade de informações de envio fornecidas através do formulário está o código postal do usuário. São projetados casos de testes para tentar-se descobrir erros no processamento do código postal especificando valores de código postal que podem revelar diversas classes de erros (por exemplo, um código postal incompleto, um código postal correto, um código postal não existente, um formato errôneo de código postal).
- *Análise de valor-limite* – Os dados de formulários são testados nos seus limites. Por exemplo, a função que calcula o frete que vimos anteriormente solicita o número máximo de dias necessários para a entrega do produto. No formulário consta um mínimo de 2 dias e um máximo de 14. Porém, o teste de valor-limite pode entrar com valores de 0, 1, 2, 13, 14 e 15 para determinar como a função reage a dados dentro ou fora dos limites de entrada válida.⁹
- *Teste de caminho* – Se a complexidade lógica da função for alta,¹⁰ pode ser usado o teste de caminho (um método de projeto de caso de teste caixa-branca) para assegurar que todos os caminhos independentes no programa foram experimentados.

Além desses métodos de projeto de casos de teste, uma técnica chamada *teste de erro forçado* [Ngu01] é usada para gerar casos de testes que propositalmente conduzem o componente da WebApp para uma condição de erro. A finalidade é descobrir erros que ocorrem durante a manipulação de erro (por exemplo, mensagens de erro incorretas ou não existentes, falha da WebApp como consequência do erro, saída errônea causada por entrada errônea, efeitos colaterais relacionados com o processamento do componente).

Cada caso de teste no nível de componente especifica todos os valores de entrada e a saída esperada a ser fornecida pelo componente. A saída real produzida decorrente do teste é registrada para referência futura durante o suporte e a manutenção.

Em muitas situações, a execução correta de uma função da WebApp está ligada ao interfaciamento correto com um banco de dados que pode ser externo a WebApp. Portanto, o teste de banco de dados torna-se parte integral do regime de teste de componente.

20.6 TESTES DE NAVEGAÇÃO

Um usuário navega por uma WebApp de maneira muito semelhante a um visitante que caminha por uma loja ou museu. Podem ser trilhados muitos caminhos, podem ser feitas muitas paradas, muitas coisas a aprender e observar, atividades a iniciar e decisões a tomar. Esse processo de navegação é previsível no sentido de que cada visitante tem uma série de objetivos quando chega. Ao mesmo tempo, o processo de navegação pode ser imprevisível porque o visitante, influenciado por alguma coisa que vê ou aprende, pode escolher um caminho ou iniciar uma ação que não é típica para o objetivo original. A tarefa do teste de navegação é (1) garantir

⁹ Nesse caso, um projeto melhor da entrada pode eliminar erros potenciais. O número máximo de dias poderia ser selecionado de um menu pull-down, impedindo o usuário de especificar entrada fora dos limites.

¹⁰ A complexidade lógica pode ser determinada computando a complexidade ciclômica do algoritmo. Veja detalhes adicionais no Capítulo 18.

que os mecanismos que permitem ao usuário navegar através da WebApp estejam todos em funcionamento e (2) confirmar que cada unidade semântica de navegação (*navigation semantic unit* — NSU) possa ser alcançada pela categoria apropriada de usuário.

20.6.1 Testando a sintaxe de navegação

A primeira fase do teste de navegação realmente começa durante o teste da interface. Os mecanismos de navegação são verificados para garantir que cada um execute sua função planejada. Splaine e Jaskiel [Spl01] sugerem os seguintes mecanismos de navegação a ser testados:

- *Links de navegação* — esses mecanismos incluem links internos dentro da WebApp, links externos para outras WebApps e âncoras dentro de uma página Web específica. Cada link deverá ser testado para assegurar que o conteúdo ou funcionalidade apropriada sejam alcançados quando o link é escolhido.
- *Redirecionamentos* — esses links entram em cena quando um usuário solicita um URL não existente ou seleciona um link cujo conteúdo foi removido ou o nome mudou. É exibida uma mensagem para o usuário, e a navegação é redirecionada para outra página (por exemplo, a página principal). Os redirecionamentos deverão ser testados solicitando-se links internos ou URLs externos incorretos e avaliando como a WebApp lida com essas solicitações.
- *Marcadores de páginas (Bookmarks)* — embora os marcadores sejam uma função do navegador, a WebApp deverá ser testada para assegurar que possa ser extraído um título de página com significado quando o marcador for criado.
- *Molduras e conjunto de molduras (Frames e framesets)* — cada moldura contém o conteúdo de uma página Web específica, e um conjunto de molduras contém múltiplas molduras e permite mostrar múltiplas páginas Web ao mesmo tempo. Por ser possível aninhar molduras e conjuntos de molduras um dentro do outro, esses mecanismos de navegação e visualização deverão ser testados quanto ao correto conteúdo, layout e dimensionamento apropriados, desempenho de download e compatibilidade de navegador.
- *Mapas de site* — o mapa de site fornece uma tabela completa de conteúdo para todas as páginas Web. Cada entrada do mapa de site deverá ser testada para garantir que os links levem o usuário ao conteúdo ou funcionalidade apropriados.
- *Dispositivos de busca interna* — WebApps complexas muitas vezes contêm centenas ou até milhares de objetos de conteúdo. Um dispositivo de busca interna permite ao usuário executar uma busca por palavra-chave dentro da WebApp para encontrar o conteúdo desejado. O teste de dispositivos de busca valida a precisão e totalidade da busca, as propriedades de manipulação de erro do dispositivo de busca e recursos avançados de busca (por exemplo, o uso de operadores booleanos no campo de busca).

Alguns dos testes citados podem ser executados por ferramentas automáticas (por exemplo, verificação de link), enquanto outros são projetados e executados manualmente. O objetivo geral é garantir que os erros em mecanismos de navegação sejam encontrados antes que a WebApp entre no ar.

20.6.2 Testando as semânticas de navegação

No Capítulo 13 a unidade semântica de navegação (NSU) é definida como “uma série de informações e estruturas de navegação relacionadas que colabora no atendimento a um subconjunto de requisitos de usuário relacionados” [Cac02]. Cada NSU é definida por um conjunto de caminhos de navegação (chamados “modos de navegação”) que conectam nós de navegação (por exemplo, páginas Web, objetos de conteúdo ou funcionalidade). Considerada como um todo, cada NSU permite ao usuário satisfazer requisitos específicos definidos por um ou mais casos de uso para uma categoria de usuário. O teste de navegação exercita cada NSU para asse-

“Não estamos perdidos. Apenas mudamos a localização.”

John M. Ford

 Quais perguntas devem ser feitas e respondidas à medida que cada NSU é testada?

gurar que esses requisitos possam ser atendidos. Você deve responder às seguintes perguntas à medida que é testada cada NSU:

- A NSU é atendida em sua totalidade sem erro?
- Cada nó de navegação (definido por uma NSU) é acessível no contexto dos caminhos de navegação definidos para a NSU?
- Se a NSU pode ser atendida usando mais de um caminho de navegação, todos os caminhos relevantes foram testados?
- Se forem fornecidas instruções pela interface de usuário para ajudar na navegação, as instruções são corretas e inteligíveis à medida que a navegação ocorre?
- Existe algum mecanismo (que não seja a seta “de retorno” do navegador) para voltar a um nó de navegação anterior e ao início do caminho de navegação?
- Os mecanismos de navegação em um nó grande de navegação (isto é, uma longa página Web) funcionam corretamente?
- Se uma função deve ser executada em um nó e o usuário opta por não fornecer entrada, o restante da NSU pode ser completado?
- Se uma função é executada em um nó e ocorre um erro no processamento da função, a NSU pode ser completada?
- Há uma maneira de interromper a navegação antes que todos os nós tenham sido alcançados, mas depois retornar ao ponto onde a navegação foi interrompida e continuar a partir dali?
- Todos os nós podem ser acessados do mapa do site? Os nomes dos nós têm significado para os usuários finais?
- Se um nó em uma NSU é alcançado de uma origem externa, é possível processar o próximo nó no caminho de navegação? É possível retornar ao nó anterior no caminho de navegação?
- O usuário entende sua localização dentro da arquitetura de conteúdo à medida que a NSU é executada?



Se não foram criadas NSUs como parte da análise ou projeto de WebApp, você pode aplicar casos de uso para o projeto de casos de testes de navegação. O mesmo conjunto de perguntas é proposto e respondido.

O teste de navegação, bem como o teste de interface e de usabilidade, deverá ser feito por diferentes clientes quando possível. Os engenheiros da Web têm a responsabilidade pelos primeiros estágios do teste de navegação, mas os estágios posteriores deverão ser testados por outros interessados, do projeto, uma equipe de teste independente e, por último, por usuários não técnicos. O objetivo é exercitar a navegação da WebApp completamente.

20.7 TESTE DE CONFIGURAÇÃO

A variabilidade e a instabilidade da configuração são fatores importantes que tornam o teste da WebApp um desafio. Hardware, sistema operacional, navegadores, capacidade de armazenamento, velocidades de comunicação de rede e uma variedade de outros fatores do lado do cliente são difíceis de prever para cada usuário. Além disso, a configuração para um usuário pode mudar regularmente [por exemplo, atualização do sistema operacional, novo provedor e novas velocidades de conexão]. O resultado pode ser um ambiente do lado do cliente, sujeito a erros sutis e significativos. A impressão de um usuário sobre a WebApp e a maneira pela qual ele interage com ela pode ser significativamente diferente da experiência de um outro usuário, se ambos não estiverem trabalhando na mesma configuração do cliente.

O objetivo do teste de configuração não é exercitar todas as configurações possíveis no lado do cliente. Em vez disso, é testar um conjunto de prováveis configurações do cliente e do servidor para assegurar que a experiência do usuário será a mesma em todos os casos e isolar erros que podem ser específicos a uma determinada configuração.

 **Quais perguntas devem ser feitas e respondidas à medida que é feito o teste de configuração no lado do servidor?**

20.7.1 Tópicos no lado do servidor

No lado do servidor, os casos de testes de configuração são projetados para verificar se a configuração do servidor [isto é, servidor WebApp, servidor de banco de dados, sistema operacional, software de firewall, aplicações concorrentes] pode suportar a WebApp sem erro. Essencialmente, a WebApp é instalada em um ambiente servidor e testado para garantir que ele opere sem erro.

Quando são projetados os testes de configuração no lado do servidor, você deve considerar cada componente da configuração. Entre as perguntas a ser feitas e respondidas durante o teste de configuração no lado do servidor destacam-se as seguintes:

- A WebApp é totalmente compatível com o sistema operacional do servidor?
- Os arquivos de sistema, diretórios e dados de sistema relacionados são criados corretamente quando a WebApp está operacional?
- As medidas de segurança do sistema (por exemplo, firewalls ou criptografia) permitem que a WebApp seja executada e sirva os usuários sem interferência ou degradação do desempenho?
- A WebApp foi testada com a configuração de servidor distribuído¹¹ (se existir alguma) que foi escolhida?
- A WebApp está adequadamente integrada com o software de banco de dados? A WebApp é sensível a diferentes versões do software de banco de dados?
- Os scripts WebApp do lado do servidor executam corretamente?
- Os erros do administrador do sistema foram examinados quanto ao seu efeito sobre as operações da WebApp?
- Se forem usados servidores substitutos (proxy), as diferenças em sua configuração foram resolvidas com o teste on-site?

20.7.2 Tópicos no lado do cliente

No lado do cliente, o teste de configuração focaliza mais intensamente a compatibilidade da WebApp com configurações que contenham uma ou mais permutações dos seguintes componentes [Ngu01]:

- *Hardware* — CPU, memória, armazenamento e dispositivos de impressão
- *Sistemas operacionais* — Linux, Macintosh OS, Microsoft Windows, sistema operacional móvel
- *Software navegador* — Firefox, Safari, Internet Explorer, Opera, Chrome, e outros
- *Componentes da interface de usuário* — Active X, Java applets e outros
- *Plug-ins* — QuickTime, RealPlayer e muitos outros
- *Conectividade* — cabo, DSL, modem regular, T1, WiFi

Além desses componentes, outras variáveis incluem software de rede, as excentricidades do provedor de serviços (ISP) e aplicações rodando concorrentemente.

Para projetar testes de configuração do cliente, deve-se reduzir o número de variáveis de configuração a um número controlável.¹² Desse modo, cada categoria de usuário é avaliada para determinar as prováveis configurações que serão encontradas. Além disso, os dados de mercado podem ser usados para prever as combinações mais prováveis de componentes. A WebApp é então testada nesses ambientes.

¹¹ Por exemplo, pode ser usado um servidor de aplicação e um servidor de banco de dados separados. A comunicação entre as duas máquinas ocorre por meio de uma conexão em rede.

¹² Rodar testes em todas as combinações possíveis de componentes de configuração é um processo extremamente demorado.

20.8 TESTE DE SEGURANÇA

“A Internet é um lugar arriscado para fazer negócios ou armazenar valores. Invasores (hackers), plagiadores (crackers), xeretas (snoops), enganadores (spoofers)... Vândalos, criadores de vírus e criadores de programas mal intencionados estão à solta.”

Dorothy e Peter Denning



Se a WebApp for crítica para os negócios, contém dados sigilosos ou é um alvo provável para invasores, é aconselhável terceirizar o teste de segurança com um prestador de serviço especializado.

Segurança de WebApp é um assunto complexo que deve ser muito bem entendido antes de se realizar um teste de segurança efetivo.¹³ WebApps e os ambientes cliente e servidor nos quais estão alojadas representam um alvo atraente para invasores (hackers) externos, funcionários insatisfeitos, concorrentes desonestos e qualquer outro que queira roubar informações sigilosas, modificar conteúdo maliciosamente, degradar o desempenho, desabilitar funcionalidade ou atrapalhar uma pessoa, organização ou negócio.

Os testes de segurança são projetados para investigar vulnerabilidades no ambiente do lado do cliente, comunicações de rede que ocorrem quando os dados são passados do cliente para o servidor e vice-versa e no ambiente do lado do servidor. Cada um desses domínios pode ser atacado, e é tarefa do testador de segurança descobrir os pontos fracos que podem ser explorados por aqueles que têm a intenção de fazer isso.

No lado do cliente, as vulnerabilidades podem ser atribuídas muitas vezes a erros preexistentes em navegadores, programas de e-mail ou software de comunicação. Nguyen [Ngu01] descreve uma brecha típica de segurança:

Um dos erros mais mencionados é o transbordamento de buffer (*buffer overflow*), que permite que código mal intencionado seja executado na máquina do cliente. Por exemplo, introduzir em um navegador um URL muito mais longo do que o tamanho de buffer alocado para o URL vai causar um erro de sobrecarga da memória (*buffer overflow*), se o navegador não tiver código de detecção de erro para validar o tamanho da entrada do URL. Um hacker experiente pode explorar esse erro escrevendo um longo URL com código a ser executado que pode fazer o navegador travar ou alterar as configurações de segurança (de alta para baixa), ou, pior ainda, corromper dados do usuário.

Outra potencial vulnerabilidade no lado do cliente é o acesso não autorizado a cookies colocados no navegador. Sites criados com intenções maliciosas podem acessar informações contidas nos cookies legítimos e usá-las para ameaçar a privacidade do usuário, ou pior, preparar o cenário para o roubo de identidade.

Dados transferidos entre cliente e servidor são vulneráveis à enganação (*spoofing*). Essa enganação ocorre quando uma extremidade do elo de comunicação é subvertida por uma entidade com intenções maliciosas. Por exemplo, um usuário pode ser enganado por um site malicioso que age como um legítimo servidor WebApp (aparência e comportamento idênticos). A intenção é roubar senhas, informações confidenciais ou dados de crédito.

No lado do servidor, as vulnerabilidades incluem ataques que causam recusa de serviço e scripts maliciosos que podem ser passados para o lado do cliente ou usados para desabilitar operações do servidor. Além disso, bancos de dados do servidor podem ser acessados sem autorização (roubo de dados).

Para a proteção contra essas vulnerabilidades (e muitas outras), é implementado um ou mais dos seguintes elementos de segurança [Ngu01]:

- *Firewal (bloqueadores contra ataques)* — mecanismo de filtragem que é uma combinação de hardware e software que examina cada pacote de informações que chega para assegurar-se de que ele esteja vindo de uma fonte legítima, bloqueando quaisquer dados suspeitos.
- *Autenticação* — mecanismo de verificação que valida a identidade de todos os clientes e servidores, permitindo que a comunicação ocorra somente quando ambos os lados são verificados.
- *Criptografia* — mecanismo de codificação que protege dados sensíveis, modificando-os de maneira que fique impossível de serem lidos por alguém com más intenções. A cripto-

13 Livros de Cross e Fisher [Cro07], Andrews e Whittaker [And06], e Trivedi [Tri03] fornecem informações úteis sobre esse assunto.

PONTO-CHAVE

Devem ser projetados testes de segurança para experimentar firewalls, autenticação, criptografia e autorização.

grafia é fortalecida pelo uso de *certificados digitais* que permitem ao cliente verificar o destino para o qual os dados são transmitidos.

- **Autorização** — mecanismo de filtragem que permite o acesso ao ambiente do cliente ou do servidor apenas por aqueles indivíduos com códigos de autorização apropriados (por exemplo, ID de usuário e senha).

Os testes de segurança deverão ser projetados para investigar cada uma dessas tecnologias de segurança em esforço para descobrir brechas na segurança.

O projeto real de testes de segurança requer profundo conhecimento do funcionamento interno de cada elemento de segurança e de uma ampla gama de tecnologias de rede. Em muitos casos, o teste de segurança é terceirizado, atribuindo-se a empresas que se especializaram nessas tecnologias.

20.9 TESTE DE DESEMPENHO

Nada é mais frustrante do que uma WebApp que leva muitos minutos para carregar o conteúdo do quando sites concorrentes fazem download de conteúdo similar em segundos. Nada é mais desgastante do que tentar entrar em uma WebApp e receber uma mensagem do tipo “servidor ocupado”, com a sugestão para você tentar mais tarde. Nada é mais desconcertante do que uma WebApp que responde instantaneamente em algumas situações e depois parece entrar em um estado de espera infinita em outras. Todas essas ocorrências acontecem diariamente na Web e todas estão relacionadas a desempenho.

O teste de desempenho é usado para descobrir problemas de desempenho que podem resultar da falta de recursos no lado do servidor, largura de banda na rede inadequada, recursos de banco de dados inadequados, recursos deficientes do sistema operacional, funcionalidade da WebApp mal projetada e outros problemas de hardware e software que podem causar degradação de desempenho cliente-servidor. A intenção é dupla: (1) entender como os sistemas respondem quando a *carga* (isto é, número de usuários, número de transações ou volume geral de dados) aumenta e (2) reunir métricas que conduzirão a modificações de projeto para melhorar o desempenho.



Alguns aspectos do desempenho da WebApp, pelo menos como são observados pelo usuário final, são difíceis de testar. A carga da rede, as excentricidades do hardware de interface de rede e problemas similares não são facilmente testados no nível da WebApp.

20.9.1 Objetivos do teste de desempenho

Os testes de desempenho são projetados para simular situações de carga do mundo real. Na medida em que cresce o número de usuários simultâneos da WebApp, ou o número de transações online aumenta, ou a quantidade de dados (download ou upload) aumenta, o teste de desempenho ajudará a responder as seguintes questões:

- O tempo de resposta do servidor degrada a um ponto em que se torna notável e inaceitável?
- Em que ponto (em termos de usuários, transações ou carga de dados) o desempenho se torna inaceitável?
- Que componentes do sistema são responsáveis pela degradação de desempenho?
- Qual o tempo médio de resposta para usuários sob uma variedade de condições de carga?
- A degradação do desempenho tem um impacto sobre a segurança do sistema?
- A confiabilidade ou precisão da WebApp é afetada quando a carga no sistema aumenta?
- O que acontece quando são aplicadas cargas maiores do que a capacidade máxima do servidor?
- A degradação de desempenho tem impacto sobre os lucros da empresa?

Para obter respostas a essas perguntas, são feitos dois testes diferentes de desempenho: (1) *teste de carga* examina cargas reais em uma variedade de níveis e em uma variedade de com-

binações, e (2) *teste de esforço (stress)* força o aumento de carga até o ponto de ruptura para determinar com que capacidade o ambiente WebApp pode lidar. Cada uma dessas estratégias é considerada a seguir.

20.9.2 Teste de carga

A finalidade do teste de carga é determinar como a WebApp e seu ambiente do lado do servidor responderá a várias condições de carga. À medida que é feito o teste, permutações das variáveis a seguir definem uma série de condições de teste:

N , número de usuários concorrentes

T , número de transações on-line por usuários por unidade de tempo

D , carga de dados processados pelo servidor por transação



Se uma WebApp usa múltiplos servidores para proporcionar uma capacidade significativa, o teste de carga deve ser feito em um ambiente multisservidor.

Em cada caso, as variáveis são definidas de acordo com os limites de operação normal do sistema. Enquanto ocorre cada uma das condições de teste, coletam-se uma ou mais das seguintes medidas: resposta média do usuário, tempo médio para o download de uma unidade padronizada de dados ou tempo médio para processar uma transação. Devem-se examinar essas medidas para determinar se uma diminuição repentina no desempenho pode ser atribuída a uma combinação específica de N , T e D .

O teste de carga também pode ser usado para avaliar velocidade de conexão recomendada para usuários da WebApp. O resultado geral, P , é calculado da seguinte maneira:

$$P = N \times T \times D$$

Como exemplo, considere um site popular de notícias esportivas. Em um momento, 20 mil usuários concorrentes enviam uma solicitação (uma transação, T) a cada 2 minutos em média. Cada transação requer que a WebApp faça o download de um novo artigo que na média tem um tamanho de 3K bytes. Portanto, o resultado pode ser calculado como:

$$\begin{aligned} P &= [20.000 \times 0,5 \times 3\text{Kb}]/60 = 500 \text{ Kbytes/segundo} \\ &= 4 \text{ megabits/segundo} \end{aligned}$$

A conexão de rede do servidor teria, portanto, de suportar essa taxa de transferência de dados e deveria ser testada para assegurar que fosse capaz disso.

20.9.3 Teste de esforço (stress)

O *teste de esforço* é uma continuação do teste de carga, mas nesse caso as variáveis, N , T e D são forçadas a alcançar e exceder os limites operacionais. A finalidade desses testes é responder às seguintes questões:

- O sistema se degrada “suavemente” ou o servidor desliga quando é excedida a capacidade?
- O software servidor gera mensagens “servidor não disponível”? De uma maneira geral, os usuários ficam cientes de que não podem acessar o servidor?
- O servidor coloca as requisições por recursos em fila e esvazia a fila quando a demanda de capacidade diminui?
- São perdidas transações quando a capacidade é excedida?
- A integridade dos dados é afetada quando a capacidade é excedida?
- Quais valores de N , T e D forçam o ambiente servidor a falhar? Como a falha se manifesta? São mandadas notificações automáticas para o pessoal de suporte técnico no local do servidor?
- Se o sistema falha, quanto tempo demora até que volte a ficar on-line?

PONTO-CHAVE

A finalidade do teste de esforço é entender melhor como o sistema falha quando é forçado além de seus limites operacionais.

- Certas funções da WebApp (por exemplo, funcionalidade de computação intensiva, recursos de encadeamento de dados) são interrompidas quando a capacidade atinge um nível de 80% ou 90%?

Uma variação do teste de esforço às vezes é chamada de teste de alternância de pico (*spike/bounce testing*) [Spl01]. Nesse regime de teste, a carga é elevada até a capacidade, depois diminuída rapidamente para as condições normais de operação e, em seguida, elevada novamente. Ao devolver a carga do sistema, você determina quão bem o servidor pode reunir recursos para atender à demanda muito alta e então liberá-los quando as condições normais se restabelecerem (de forma que fiquem prontos para o próximo pico).

20.10 RESUMO

O objetivo do teste de WebApp é experimentar cada uma das muitas dimensões da qualidade da WebApp com a finalidade de encontrar erros ou descobrir problemas que podem levar a falhas de qualidade. O teste focaliza o conteúdo, função, estrutura, utilização, navegabilidade, desempenho, compatibilidade, interoperabilidade, capacidade e segurança. Incorpora revisões que ocorrem quando a WebApp é projetada e testes feitos depois que a WebApp é implantada.

A estratégia de teste para WebApp experimenta cada dimensão da qualidade examinando inicialmente “unidades” de conteúdo, funcionalidade ou navegação. Uma vez validadas as unidades individuais, o foco passa para os testes que experimentam a WebApp como um todo. Para tanto, são criados muitos testes sob a perspectiva do usuário e controlados por informações contidas em casos de uso. Um plano de teste para WebApp é desenvolvido e identifica as etapas

FERRAMENTAS DO SOFTWARE**Taxonomia de ferramentas para teste de WebApps**

Em seu artigo sobre teste de sistemas de e-commerce, Lam [Lam01] apresenta uma taxonomia útil de ferramentas automáticas com aplicabilidade direta para teste em um contexto de engenharia para Web. Acrescentamos ferramentas representativas em cada uma das categorias.¹⁴

Ferramentas de configuração e gerenciamento de conteúdo gerenciam versão e controle de alterações de objetos de conteúdo e componentes funcionais da WebApp.

Ferramentas representativas:

Uma lista abrangente está em www.daveeaton.com/scm/CMTools.html

Ferramentas de desempenho de banco de dados medem desempenho de banco de dados, como, por exemplo, o tempo necessário para executar consultas selecionadas a banco de dados. Essas ferramentas facilitam a otimização do banco de dados.

Ferramentas representativas:

BMC Software (www.bmc.com)

Depuradores são ferramentas típicas de programação que localizam e resolvem defeitos de software no código. Fazem parte da maioria dos ambientes modernos de desenvolvimento de aplicações.

Ferramentas representativas:

Accelerated Technology (www.acceleratedtechnology.com)

Apple Debugging Tools (developer.apple.com/tools/performance/)

IBM VisualAge Environment (www.ibm.com)

Microsoft Debugging Tools (www.microsoft.com)

Sistemas de controle de defeitos registram defeitos e rastreiam seu estado e solução. Algumas incluem ferramentas de relatórios para fornecer informações de gerenciamento sobre taxas de propagação de defeitos e de solução de defeitos.

Ferramentas representativas:

EXCEL Quickbigs (www.excelsoftware.com)

ForeSoft BugTrack (www.bugtrack.net)

McCabe TRUETrack (www.mccabe.com)

Ferramentas de monitoramento de rede observam o nível de tráfego na rede. São úteis para identificar gargalos na rede e teste de link entre sistemas frontal e de retaguarda.

Ferramentas representativas:

Lista em www.slac.stanford.edu/xorg/nmtf/nmtftools.html

Ferramentas de teste de regressão armazenam casos de testes e dados de teste e podem reaplicar os casos de testes após sucessivas mudanças de software.

¹⁴ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos revendedores.

Ferramentas representativas:

Compuware QARun (www.compuware.com/products/qacenter/qarun)

Rational VisualTest (www.rational.com)

Seque Software (www.seque.com)

Ferramentas de monitoramento de site monitora o desempenho de um site, muitas vezes, sob a perspectiva de usuário. Use-as para compilar estatísticas como, por exemplo, tempo de resposta de um extremo ao outro (end-to-end) e vazão (*throughput*) e para verificar periodicamente a disponibilidade de um site.

Ferramentas representativas:

Keynote Systems (www.keynote.com)

Ferramentas de esforço ajudam os desenvolvedores a explorar o comportamento do sistema sob altos níveis de uso operacional e a localizar os pontos de ruptura de um sistema.

Ferramentas representativas:

Mercury Interactive (www.merc-int.com)

Open-source testing tools (www.opensourcetesting.org/performance.php)

Web Performance Load Tester (www.webperformanceinc.com)

Monitores de recurso de sistema fazem parte de muitos sistemas operacionais de servidor e software servidor Web; eles monitoram recursos como espaço de disco, uso da CPU e memória.

Ferramentas representativas:

Successful Hosting.com (www.successfulhosting.com)

Quest Software Foglight (www.quest.com)

Ferramentas de geração de dados de teste ajudam os usuários a gerar dados de teste.

Ferramentas representativas:

Lista em www.softwareqatest.com/qatweb1.html

Comparadores de resultados de teste ajudam a comparar os resultados de um conjunto de testes com os resultados de outro conjunto. Use-as para verificar se as alterações no código não introduziram alterações adversas no comportamento do sistema.

Ferramentas representativas:

Lista útil em www.aptest.com/resources.html

Monitores de transação medem o desempenho de sistemas de processamento de alto volume de transações.

Ferramentas representativas:

QuotiumPro (www.quotium.com)

Software Research eValid (www.soft.com/eValid/index.html)

Ferramentas de segurança de site ajudam a detectar problemas potenciais de segurança. Pode-se muitas vezes configurar ferramentas de investigação de segurança e monitoramento para rodar em horários programados.

Ferramentas representativas:

Lista em www.timberlinetechnologies.com/products/www.html

do teste, os artefatos finais (por exemplo, casos de teste) e os mecanismos para a avaliação dos resultados. O processo de teste abrange sete diferentes tipos de teste.

Teste de conteúdo (e revisões) concentra-se nas várias categorias de conteúdo. O objetivo é descobrir erros de sintaxe e semântica que afetam a exatidão do conteúdo ou a maneira pela qual ele é apresentado ao usuário final. O teste de interface exercita os mecanismos de interação que possibilitam a um usuário se comunicar com a WebApp e validar aspectos estéticos da interface. A finalidade é descobrir erros que resultam de mecanismos de interação mal-implementados ou de omissões, inconsistências ou ambiguidades em semânticas de interface.

O teste de navegação aplica casos de uso criados como parte da atividade de modelagem, no projeto de casos de testes que experimentam cada cenário de uso em relação ao projeto de navegação. Os mecanismos de navegação são verificados para assegurar que quaisquer erros que impedem a realização de um caso de uso sejam identificados e corrigidos. O teste de componente experimenta as unidades de conteúdo e funcional da WebApp.

O teste de configuração tenta descobrir erros e/ou problemas de compatibilidade específicos de um ambiente particular de cliente ou servidor. São então aplicados testes para descobrir erros associados a cada configuração possível. O teste de segurança incorpora uma série de testes projetados para explorar vulnerabilidades na WebApp e seu ambiente. A finalidade é encontrar brechas de segurança. O teste de desempenho abrange uma série de testes projetados para avaliar o tempo de resposta e a confiabilidade da WebApp quando aumentam as demandas de recursos do servidor.

PROBLEMAS E PONTOS A PONDERAR

20.1. Existem quaisquer situações nas quais o teste de WebApp deveria ser totalmente desconsiderado?

20.2. Discuta os objetivos do teste em um contexto de WebApp.

- 20.3.** Compatibilidade é uma dimensão de qualidade importante. O que deve ser testado para garantir que exista compatibilidade em uma WebApp?
- 20.4.** Quais erros tendem a ser mais sérios — erros no cliente ou erros no servidor? Por quê?
- 20.5.** Quais elementos da WebApp podem ser “testados em unidade”? Que tipos de testes devem ser executados apenas depois que os elementos da WebApp estiverem integrados?
- 20.6.** É sempre necessário desenvolver um plano formal de teste escrito? Explique.
- 20.7.** É correto dizer que a estratégia geral de teste para WebApp começa com elementos visíveis ao usuário e passa para os elementos de tecnologia? Há exceções a essa estratégia?
- 20.8.** O teste de conteúdo está *realmente* testando em um sentido convencional? Explique.
- 20.9.** Descreva os passos associados ao teste de banco de dados para uma WebApp. O teste de banco de dados é predominantemente uma atividade do lado do cliente ou do lado do servidor?
- 20.10.** Qual a diferença entre teste que está associado a mecanismos de interface e teste que cuida de semânticas de interface?
- 20.11.** Suponha que você esteja desenvolvendo uma farmácia on-line (**YouCornerPharmacy.com**) dedicada aos aposentados e idosos. A farmácia tem funções típicas, mas também mantém uma base de dados de cada cliente para que possa fornecer informações sobre medicamentos e alertar sobre interações de certos medicamentos. Discuta quaisquer testes especiais de utilização para essa WebApp.
- 20.12.** Suponha que você tenha implementado uma função de verificação de interação de medicamentos para a (Problema 20.11). Discuta os tipos de testes no nível de componente que teriam de ser feitos para garantir que essa função funcione corretamente. [Nota: Teria de ser usada uma base de dados para implementar essa função.]
- 20.13.** Qual a diferença entre teste de sintaxe de navegação e teste de semânticas de navegação?
- 20.14.** É possível testar todas as configurações que uma WebApp pode encontrar no lado do servidor? E no lado do cliente? Se não, como você seleciona um conjunto significativo de testes de configuração?
- 20.15.** Qual o objetivo do teste de segurança? Quem executa essa atividade de teste?
- 20.16.** A **YouCornerPharmacy.com** (Problema 20.11) tornou-se extraordinariamente bem-sucedida, e o número de usuários aumentou significativamente nos primeiros dois meses de operação. Trace um gráfico que mostra o tempo de resposta provável em função do número de usuários para um conjunto fixo de recursos no servidor. Inclua legendas no gráfico para indicar pontos de interesse na “curva de resposta”.
- 20.17.** Em resposta ao seu sucesso, a **YouCornerPharmacy.com** (Problema 20.11) implementou um servidor especial para cuidar de refis (novas doses de medicamentos) das receitas. Na média, 1.000 usuários concorrentes enviam uma solicitação de refil a cada dois minutos. A WebApp faz o download de um bloco de 500 bytes de dados em resposta. Qual a vazão/taxa de saída (*throughput*) aproximada necessária para esse servidor em megabits por segundo?
- 20.18.** Qual a diferença entre teste de carga e teste de esforço?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A literatura para teste de WebApp continua a evoluir. Livros de Andrews e Whittaker (*How to Break Web Software*, Addison-Wesley, 2006), Ash (*The Web Testing Companion*, Wiley, 2003), Nguyen and his colleagues (*Testing Applications for the Web*, 2d ed., Wiley, 2003), Dustin and his colleagues (*Quality Web Systems*, Addison-Wesley, 2002), e Splaine and Jaskiel [Spl01] estão entre os tratados mais completos do assunto publicado até hoje. Mosley (*Client-Server*

Software Testing on the Desktop and the Web, Prentice Hall, 1999) examina os problemas de teste tanto no lado do cliente quanto no do servidor.

Informações úteis sobre estratégias de teste de WebApp e métodos, bem como uma discussão conveniente sobre ferramentas de teste automáticas, são apresentadas por Stottlemeyer (*Automated Web Testing Toolkit*, Wiley, 2001). Graham and her colleagues (*Software Test Automation*, Addison-Wesley, 1999) oferecem material adicional sobre ferramentas automáticas.

Microsoft (*Performance Testing Guidance for Web Applications*, Microsoft Press, 2008) e Subraya (*Integrated Approach to Web Performance Testing*, IRM Press, 2006) apresentam tratamentos detalhados sobre teste de desempenho para WebApps. Chirillo (*Hack Attacks Revealed*, 2d ed., Wiley, 2003), Splaine (*Testing Web Security*, Wiley, 2002), Klevinsky e colegas (*Hack I.T.*):

Security through Penetration Testing, Addison-Wesley, 2002), e Skoudis (*Counter Hack*, Prentice Hall, 2001) proporcionam muitas informações úteis para aqueles que devem projetar testes de segurança. Além disso, livros que tratam de teste de segurança para software em geral podem oferecer orientações importantes para aqueles que devem testar WebApps. Entre os títulos representativos estão: Basta and Halton (*Computer Security and Penetration Testing*, Thomson Delmar Learning, 2007), Wysopal e colegas (*The Art of Software Security Testing*, Addison-Wesley, 2006), e Gallagher e colegas (*Hunting Security Bugs*, Microsoft Press, 2006).

Há uma grande variedade de recursos de informação sobre teste de WebApp disponível na Internet. Uma lista atualizada de referências na Web, relevante ao teste de WebApp, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser/htm**.

MODELAGEM FORMAL E VERIFICAÇÃO

CONCEITOS- -CHAVE

certificação	499
especificação de	
estrutura de caixa . .	492
especificação	
funcional.	493
linguagem de	
especificação Z . . .	508
linguagens de	
especificações	
formais.	504
modelo de processo	
sala limpa.	493
object constraint	
language (OCL) . . .	505
projeto sala limpa .	496
refinamento do	
projeto.	496
verificação	
de correção.	492

Diferentemente das revisões e testes que começam logo que os modelos e os códigos de software são desenvolvidos, a modelagem formal e a verificação incorporam métodos de modelagem especializados que são integrados com abordagens prescritas de verificação. Sem a abordagem apropriada, a verificação não pode ser feita.

Neste capítulo, discutem-se dois métodos formais de modelagem e verificação — *engenharia de software sala limpa* e *métodos formais*. Ambos demandam abordagem de verificação especializada e a cada um se aplica um método único de verificação. Ambos são muito rigorosos e nenhum deles é usado amplamente pela comunidade de engenharia de software. Mas para criar um software “à prova de balas”, esses métodos podem ser muitos úteis. Vale a pena conhecê-los.

Engenharia de software sala limpa é uma abordagem que enfatiza a necessidade de agregar precisão ao software enquanto está sendo desenvolvido. Em vez do ciclo clássico de análise, codificação, teste e depuração, a abordagem sala limpa sugere um diferente ponto de vista [Lin94b]:

A filosofia por trás da engenharia de software sala limpa é evitar a dependência de processos onerosos de remoção de defeitos, escrevendo incrementos de código corretos já na primeira vez e verificando sua precisão antes do teste. Seu modelo de processo incorpora a certificação estatística de qualidade dos incrementos de código à medida que vão se acumulando no sistema.

PANORAMA

O que é? Quantas vezes você já ouviu alguém dizer, “Faça direito já na primeira vez”? Se aplicássemos isso ao software, seriam gastos bem menos esforços com retrabalho desnecessário de software. Dois métodos avançados de engenharia de software — engenharia de software sala limpa e métodos formais — ajudam a equipe de software a “fazer direito já na primeira vez” proporcionando uma abordagem matemática baseada na modelagem de programa e na habilidade de verificar se o modelo está correto. A engenharia de software de sala limpa enfatiza a verificação matemática da correção antes de começar a construção do programa e a certificação da confiabilidade do software como parte da atividade de teste. Os métodos formais usam a teoria dos conjuntos e a notação lógica para criar uma definição clara dos fatos (requisitos) que podem ser analisados para melhorar (ou mesmo provar) a correção e consistência. O limite inferior para ambos os métodos é a criação de software com taxas de falhas extremamente baixas.

Quem realiza? Um engenheiro de software especialmente treinado.

Por que é importante? Erros criam retrabalho. O retrabalho consome tempo e aumenta os custos. Não seria maravilhoso se pudéssemos reduzir significativamente a quantidade de erros (defei-

tos, bugs) introduzidos à medida que o software é projetado e construído? Essa é a premissa da modelagem e verificação formais.

Quais são as etapas envolvidas? Modelos de requisitos e de projeto são criados usando notação especializada que é favorável à verificação matemática. A engenharia de software de sala limpa utiliza representação de estrutura de caixas que encapsula o sistema (ou algum aspecto do sistema) em um nível específico de abstração. É aplicada a verificação da correção uma vez completo o projeto de estrutura de caixas. Depois de verificada a correção para cada estrutura de caixas, inicia-se o teste de uso estatístico. Métodos formais traduzem os requisitos do software para uma representação mais formal aplicando a notação e heurística de conjuntos para definir a invariante dos dados, estados e operações para uma função do sistema.

Qual é o artefato? É desenvolvido um modelo especializado e formal de requisitos. São registrados os resultados das provas de correção e dos testes estatísticos de uso.

Como garantir que o trabalho foi realizado corretamente? É aplicada uma prova formal de correção ao modelo de requisitos. Os testes estatísticos de uso experimentam cenários de uso para assegurar que os erros na funcionalidade de usuário sejam descobertos e corrigidos.

De muitas formas, a abordagem sala limpa eleva a engenharia de software a outro nível, enfatizando a necessidade de *provar* a precisão.

Modelos desenvolvidos por meio de *métodos formais* são descritos usando sintaxe e semântica formais que especificam a função e o comportamento do sistema. A especificação é na forma matemática (por exemplo, o cálculo de predicado pode ser usado como base para uma linguagem de especificação formal). Em sua introdução aos métodos formais, Anthony Hall [Hal90] faz um comentário que se aplica igualmente aos métodos sala limpa:

Métodos formais [e a engenharia de software sala limpa] são controversos. Seus defensores dizem que podem revolucionar o desenvolvimento [de software]. Seus críticos acham que são impossivelmente difíceis. Enquanto isso, para a maioria das pessoas, métodos formais [e engenharia de software sala limpa] são tão estranhos que é difícil julgar os argumentos a favor ou contra.

Neste capítulo, exploraremos os métodos formais de modelagem e verificação e examinaremos seu impacto sobre a engenharia de software nos próximos anos.

21.1 ESTRATÉGIA SALA LIMPA

“A única maneira de os erros ocorrerem em um programa é serem colocados lá pelo autor. Não há outros mecanismos conhecidos... A prática correta procura evitar a inserção de erros e, se isso falhar, deve-se removê-los antes de testar ou de utilizar qualquer outro programa.”

Harlan Mills

WebRef

Uma excelente fonte de informações e recursos para engenharia de software sala limpa pode ser encontrada em www.cleansoft.com.

“A engenharia de software sala limpa consegue um controle estatístico de qualidade sobre o desenvolvimento de software separando estritamente o processo de projeto do processo de teste em uma sequência de desenvolvimento incremental de software.”

Harlan Mills

A engenharia de software sala limpa usa uma versão especializada do modelo incremental de software introduzido no Capítulo 2. Uma “sequência de incrementos de software” [Lin94b] é desenvolvida por pequenas equipes de software independentes. À medida que cada incremento é certificado, ele é integrado no todo. Daí o motivo pelo qual o sistema cresce com o tempo.

A sequência das tarefas sala limpa para cada incremento está ilustrada na Figura 21.1. De acordo com a sequência para incrementos sala limpa, ocorrem as seguintes tarefas:

Planejamento de incremento. É desenvolvido um plano de projeto que adota a estratégia incremental. São criados a funcionalidade de cada incremento, seu tamanho projetado e um cronograma de desenvolvimento sala limpa. Deve-se ter cuidado especial para garantir que os incrementos certificados serão integrados no seu devido tempo.

Coleta dos requisitos. Usando técnicas similares às aquelas introduzidas no Capítulo 5, é desenvolvida uma descrição mais detalhada dos requisitos no nível do cliente (para cada incremento).

Especificação da estrutura de caixa. É usado um método de especificação que emprega *estruturas de caixas* para descrever a especificação funcional. As estruturas de caixas “isolam e separam a definição criativa de comportamento, dados e procedimentos em cada nível de refinamento” [Hev93].

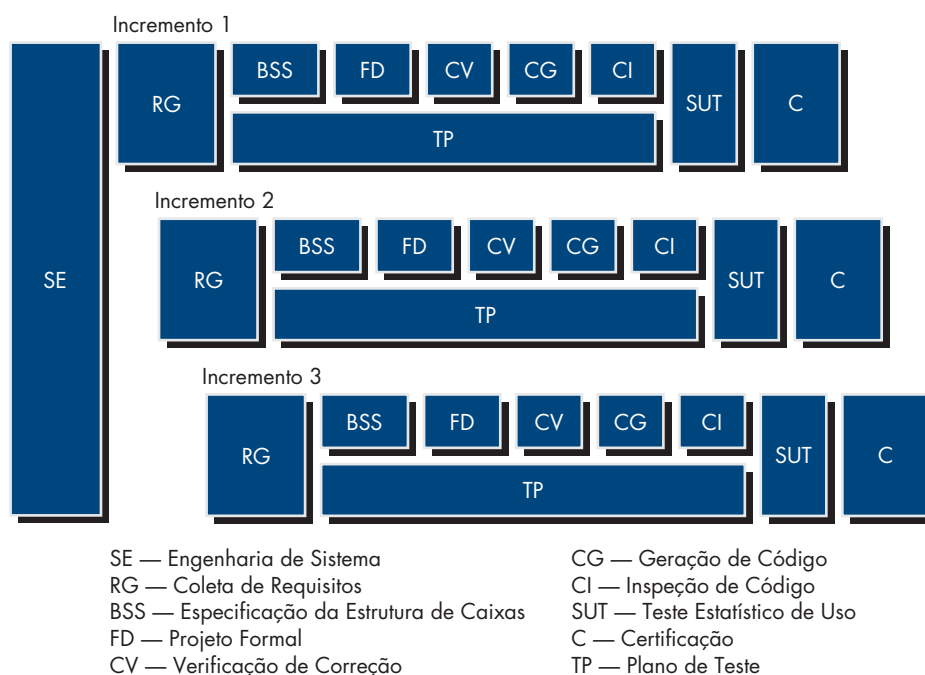
Projeto formal. Por meio da abordagem de estrutura de caixas, o projeto sala limpa é uma extensão natural e contínua da especificação. Embora seja possível fazer uma clara distinção entre as duas atividades, as especificações (chamadas de *caixas-pretas*) são refinadas iterativamente (dentro de um incremento) para se tornarem análogas aos projetos arquitetônicos e no nível de componentes (chamados de *caixas de estado* e *caixas-claras*, respectivamente).

Verificação de correção. A equipe sala limpa executa uma série de atividades de verificação rigorosa da correção sobre o projeto e depois sobre o código. A verificação (Seção 21.3.2) começa com a estrutura de caixas de nível mais alto (especificação) e move-se em direção ao detalhe de projeto e código. O primeiro nível de verificação de correção ocorre aplicando-se uma série de “perguntas de correção” [Lin88]. Se essas perguntas não demonstrarem que a especificação está correta, utilizam-se métodos de verificação mais formais (matemáticos).

Geração de código, inspeção e verificação. As especificações de estrutura de caixa, representadas em uma linguagem especializada, são traduzidas em linguagem de programação apropriada. São usadas revisões técnicas (Capítulo 15) para garantir a conformidade

FIGURA 21.1

O modelo de processo sala limpa



da semântica do código e das estruturas de caixa e correção sintática do código. Então é executada a verificação de correção para o código fonte.



A sala limpa concentra-se em testes que experimentam a maneira como o software é realmente usado. Casos de uso fornecem informações para o processo de planejamento de teste.

Planejamento do teste estatístico. É analisado o uso projetado do software, e é planejada e projetada uma série de casos de teste que experimentam uma “distribuição de probabilidades” de uso (Seção 21.4). De acordo com a Figura 21.1, essa atividade sala limpa é executada em paralelo à especificação, verificação e geração de código.

Teste estatístico de uso. Recordando que o teste exaustivo de software é algo impossível (Capítulo 18), é sempre necessário projetar um número finito de casos de teste. As técnicas estatísticas de uso [Poo88] executam uma série de testes derivados de uma amostragem estatística (a distribuição de probabilidades que mencionamos antes) de todas as execuções de programa possíveis por todos os usuários de uma população-alvo (Seção 21.4).

Certificação. Uma vez completadas a verificação, inspeção e teste de uso (e todos os erros forem corrigidos), o incremento é certificado como pronto para a integração.

As quatro primeiras atividades no processo sala limpa preparam o cenário para as atividades de verificação formal que vêm em seguida. Por essa razão, começa-se a discussão da abordagem sala limpa com as atividades de modelagem que são essenciais para a verificação formal a ser aplicada.

21.2 ESPECIFICAÇÃO FUNCIONAL

“Há uma coisa curiosa sobre a vida: se você se recusa a aceitar qualquer coisa que não seja o melhor, frequentemente acabará obtendo o melhor.”

W. Somerset Maugham

A abordagem de modelagem em engenharia de software sala limpa usa um método chamado de *especificação de estrutura de caixa*. Uma “caixa” encapsula o sistema (ou algum aspecto do sistema) em algum nível de detalhe. Por meio de um processo de elaboração ou refinamento passo a passo, as caixas são refinadas em uma hierarquia em que cada caixa tem transparência referencial. Isto é, “O conteúdo de informações da especificação de cada caixa é suficiente para definir seu refinamento, sem depender da implementação de qualquer outra caixa” [Lin94b]. Isso habilita o analista a particionar um sistema hierarquicamente, mudando da representação essencial no topo para o detalhe específico de implementação na base. São usados três tipos de caixas:

? Como se consegue um refinamento como parte de uma especificação de estrutura de caixa?

Caixa-preta. A caixa-preta especifica o comportamento de um sistema ou uma parte de um sistema. O sistema (ou parte) responde a estímulos específicos (eventos) aplicando um conjunto de regras de transição que mapeiam os estímulos em uma resposta.

Caixa de estado. A caixa de estado encapsula dados de estado e serviços (operações) de maneira análoga a objetos. Nessa visão de especificação, são representadas entradas para a caixa de estado (estímulos) e saídas (respostas). A caixa de estado representa o “histórico de estímulo” da caixa-preta, isto é, os dados encapsulados na caixa de estado que devem ser retidos entre as transições inferidas.

Caixa-clara. As funções de transição inferidas pela caixa de estado são definidas na caixa-clara. Em outras palavras, uma caixa-clara contém o projeto procedimental para a caixa de estado.

A Figura 21.2 ilustra a abordagem de refinamento usando a especificação de estrutura de caixas. Uma caixa-preta (BB1) define as respostas para um conjunto completo de estímulos. BB1 pode ser refinado em um conjunto de caixas-pretas, BB1.1 até BB1.n, cada uma das quais lida com uma classe de comportamento. O refinamento continua até que seja identificada uma classe coerente de comportamento (por exemplo, BB1.1.1). Uma caixa de estado (SB1.1.1) é então definida para a caixa-preta (BB1.1.1). Nesse caso, SB1.1.1 contém todos os dados e serviços necessários para implementar o comportamento definido por BB1.1.1. Por fim, SB1.1.1 é refinada em caixas-claras (CB1.1.1.1.n) e são especificados os detalhes do projeto procedimental.

À medida que ocorre cada um desses passos de refinamento, ocorre também a verificação de correção. As especificações de caixa de estado são verificadas para garantir que cada uma esteja conforme o comportamento definido pela especificação da caixa-preta pai. De maneira semelhante, as especificações caixas-claras são verificadas em relação à caixa de estado pai.

FIGURA 21.2

Refinamento da estrutura de caixas

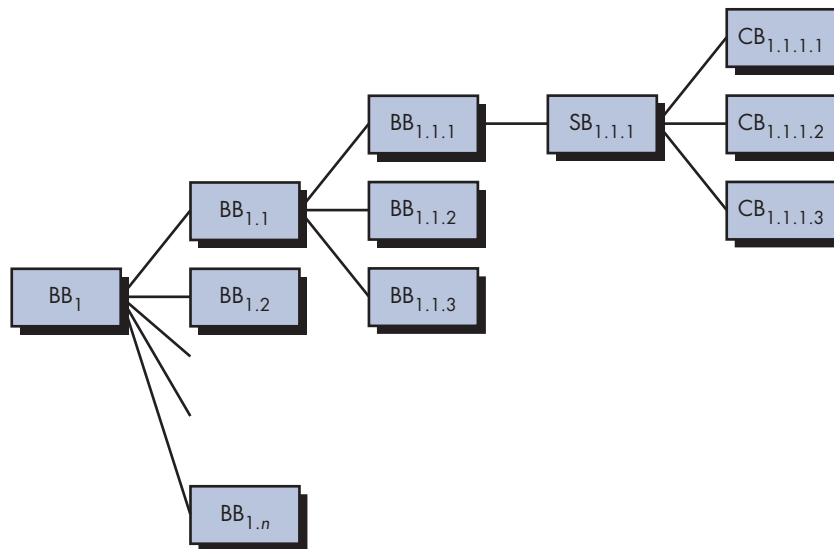
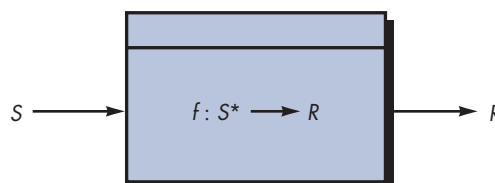


FIGURA 21.3

Uma especificação caixa-preta



21.2.1 Especificação caixa preta

A especificação *caixa-preta* descreve uma abstração, estímulos e respostas usando a notação mostrada na Figura 21.3 [Mil88]. A função f é aplicada a sequência S^* de entradas (estímulos) S e as transforma em uma saída (resposta) R . Para componentes simples de software, f pode ser uma função matemática, mas em geral, f é descrita por meio de linguagem natural (ou linguagem de especificação formal).

Muitos dos conceitos introduzidos para sistemas orientados a objetos são também aplicáveis à caixa-preta. As abstrações de dados e as operações que manipulam as abstrações são encapsuladas pela caixa-preta. Como em hierarquia de classe, a especificação caixa-preta pode exibir hierarquias de uso nas quais as caixas de baixo nível herdam as propriedades daquelas caixas de mais alto nível na estrutura.

21.2.2 Especificação caixa de estado

A *caixa de estado* é uma simples generalização de uma máquina de estado [Mil88]. Lembrando da discussão da modelagem comportamental e diagramas de estado no Capítulo 7, um estado é algum modo observável do comportamento de um sistema. À medida que ocorre o processamento, um sistema responde a eventos (estímulos) fazendo uma transição do estado atual para algum novo estado. Enquanto é feita a transição, pode ocorrer uma ação. A caixa de estado utiliza a abstração de dados para determinar a transição para o próximo estado e a ação (resposta) que ocorrerá em consequência da transição.

Observando a Figura 21.4, a caixa de estado incorpora uma caixa-preta g . O estímulo S , colocado na caixa-preta, chega de alguma fonte externa e de um conjunto de estados internos de sistema T . Mills [Mil88] fornece uma descrição matemática da função f da caixa-preta contida na caixa de estado:

$$g : S^* \times T^* \rightarrow R \times T$$

em que g é uma subfunção ligada a um estado específico t . Quando considerada coletivamente, os pares estado-subfunção (t, g) definem a função caixa-preta f .

21.2.3 Especificação caixa clara

A especificação caixa-clara está estreitamente relacionada com o projeto procedimental e programação estruturada. Essencialmente, a subfunção g dentro da caixa de estado é substituída pelas construções de programação estruturada que implementam g .

Como exemplo, considere a caixa-clara da Figura 21.5. A caixa-preta g , da Figura 21.3, é substituída por uma sequência de construções que incorpora uma condicional.

FIGURA 21.4
Especificação
caixa-preta

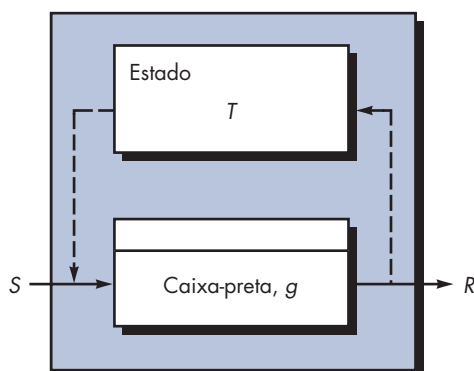
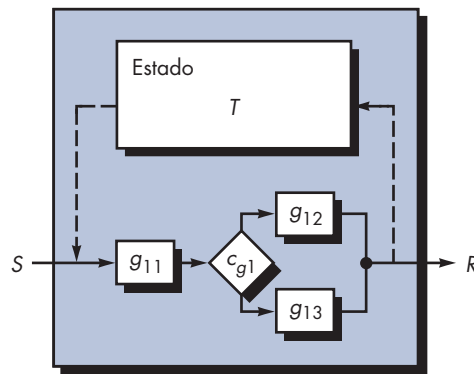


FIGURA 21.5
Especificação
caixa-clara



Estas, por sua vez, podem ser refinadas em caixas-claras em um nível mais baixo, à medida que prossegue o refinamento passo a passo.

É importante notar que a especificação procedimental descrita na hierarquia caixa-clara pode ter sua correção provada. Esse tópico é considerado na Seção 21.3.

21.3 PROJETO SALA LIMPA

A engenharia de software sala limpa usa intensamente a filosofia de programação estruturada (Capítulo 10). Mas neste caso, a programação estruturada é aplicada com muito mais rigor.

Funções básicas de processamento (descritas durante os primeiros refinamentos da especificação) são refinadas usando uma “expansão passo a passo de funções matemáticas em estruturas de conectivos lógicos [por exemplo, if-then-else] e subfunções, em que a expansão [é] executada até que todas as subfunções identificadas possam ser diretamente declaradas na linguagem de programação empregada para implementação” [Dye92].

A abordagem de programação estruturada pode ser usada eficazmente para refinar uma função, mas e o projeto de dados? Aqui entram em cena muitos conceitos fundamentais de projeto (Capítulo 8). Dados de programação são encapsulados como um conjunto de abstrações atendidas por subfunções. Utilizam-se os conceitos de encapsulamento de dados, ocultamento de informações e tipos de dados para criar o projeto de dados.

21.3.1 Refinamento de projeto

Cada especificação caixa-clara representa o projeto de um procedimento (subfunção) necessário para obter uma transição caixa de estado. Na caixa-clara, construções de programação estruturada e refinamento passo a passo são usados para representar o detalhe procedimental. Por exemplo, uma função de programa f é refinada em uma sequência de subfunções g e h . Estas, por sua vez, são refinadas em construções condicionais (por exemplo, if-then-else e do-while). O refinamento continua até que haja detalhe procedimental suficiente para criar o componente em questão.

Em cada nível de refinamento, a equipe sala limpa¹ executa uma *verificação formal de correção*. Para tanto, uma série de condições genéricas de correção é anexada às construções de programação estruturada. Se uma função f é expandida em uma sequência g e h , a condição de precisão para toda a entrada a é

- g seguido por h faz f ?

¹ Como a equipe inteira está envolvida no processo de verificação, é pouco provável que seja produzido um erro na condição da própria verificação.

Que condições são aplicadas para provar a correção das construções estruturadas?

Quando uma função p é refinada para uma condicional da forma, $\text{if } \langle c \rangle \text{ then } q, \text{ else } r$, a condição de correção para toda entrada em p é

- Sempre que a condição $\langle c \rangle$ for verdadeira, q faz p ; e sempre que $\langle c \rangle$ for falsa, r faz p ?

Quando a função m é refinada como um laço, as condições de correção para toda a entrada a m são

- O término está garantido?
- Sempre que $\langle c \rangle$ for verdadeiro, n seguido de m faz m ; e sempre que $\langle c \rangle$ for falsa, a saída do ciclo ainda faz m ?

A cada vez que uma caixa-clara for refinada para o próximo nível de detalhe, essas condições de correção são aplicadas.

21.3.2 Verificação de projeto

Você deve observar que o uso de construções de programação estruturada limita o número de testes de correção que devem ser feitos. É verificada uma única condição para as sequências; são testadas duas condições para if-then-else e verificadas três condições para laços.

Para ilustrarmos a verificação de correção para um projeto procedimental, usamos um exemplo simples introduzido por Linger, Mills e Witt [Lin79]. O objetivo é projetar e verificar um pequeno programa que encontra a parte inteira y de uma raiz quadrada de um número inteiro x . O projeto procedimental é representado usando-se o fluxograma da Figura 21.6.²

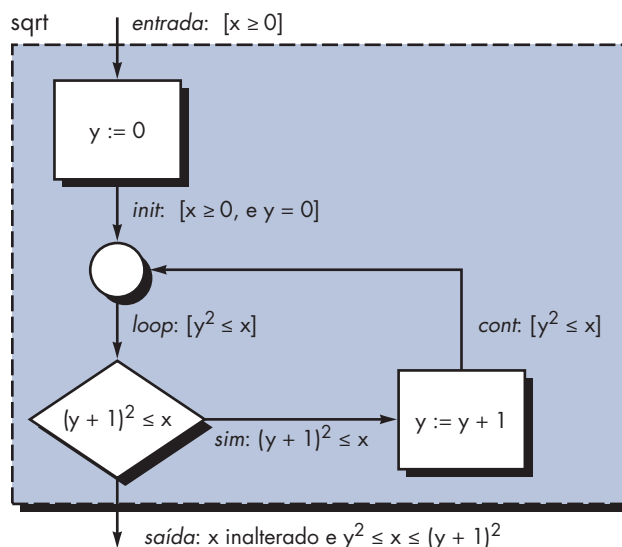
Para verificar a correção desse projeto, são acrescentadas condições de entrada e saída conforme mostra a Figura 21.6. A condição de entrada menciona que x deve ser maior ou igual a 0. A condição de saída requer que x permaneça inalterado e que y satisfaça a expressão da Figura. Para provar que o projeto está correto, é necessário provar que as condições *init*, *loop*, *cont*, *sim* e *saída* mostradas na Figura 21.6 são verdadeiras em todos os casos. Às vezes chama-se isso de subprovas.

1. A condição *init* demanda que $[x \geq 0 \text{ e } y = 0]$. Baseado nos requisitos do problema, assume-se que a condição de entrada é correta.³ Portanto, a primeira parte da condição *init*, $x \geq 0$, é satisfeita. Observando o fluxograma, a instrução que precede imediatamente a condição *init* faz $y = 0$. Portanto, a segunda parte da condição *init* também é satisfeita. Daí, *init* é verdadeira.

FIGURA 21.6

Calculando a parte inteira de uma raiz quadrada

Fonte: (Lin79)



² A Figura 21.6 foi adaptada de [Lin94]. Usada com permissão.

³ Um valor negativo para a raiz quadrada não tem significado nesse contexto.

PONTO-CHAVE

Para provar que um projeto está correto, você precisa primeiro identificar todas as condições e então provar que cada uma delas assume um valor booleano apropriado. Elas são chamadas de *subprovas*.

2. A condição *loop* pode ser encontrada em uma de duas formas: (1) diretamente de *init* (neste caso, a condição *loop* é satisfeita diretamente) ou via fluxo de controle que passa através da condição *cont*. Como a condição *cont* é idêntica à condição *loop*, *loop* é verdadeira independentemente do caminho que leva até ela.
3. A condição *cont* é encontrada apenas depois que o valor de y é incrementado de 1. Além disso, o caminho do fluxo de controle que leva a *cont* pode ser invocado somente se a condição *sim* também for verdadeira. Daí, se $(y + 1)^2 \leq x$, segue-se que $y^2 \leq x$. A condição *cont* é satisfeita.
4. A condição *sim* é testada na lógica condicional mostrada. Logo, a condição *sim* deve ser verdadeira quando o fluxo de controle se move ao longo do caminho mostrado.
5. A condição *saída* primeiro demanda que x permaneça inalterado. Um exame do projeto indica que x não aparece em nenhum lugar à esquerda de um operador de atribuição. Não há chamadas de função que use x . Portanto, ele fica inalterado. Como o teste condicional $(y + 1)^2 \leq x$ deve falhar em atingir a condição de *saída*, segue-se que $(y + 1)^2 \leq x$. Além disso, a condição *loop* deve ainda ser verdadeira (isto é, $y^2 \leq x$). Portanto, $(y + 1)^2 > x$ e $y^2 \leq x$ podem ser combinados para satisfazer a condição de saída.

Você deve ainda garantir que o laço termine. Um exame da condição *loop* indica que, porque y é incrementado e $x \geq 0$, o laço deve eventualmente terminar.

Os cinco passos que acabamos de descrever são uma prova da correção do projeto do algoritmo mostrado na Figura 21.6. Você tem certeza agora que o projeto irá, de fato, calcular a parte inteira de uma raiz quadrada.

É possível uma abordagem matemática mais rigorosa da verificação de projeto. No entanto, uma discussão desse tópico está além do escopo deste livro. Se você tiver interesse, consulte [Lin79].

21.4 TESTE SALA LIMPA

“A qualidade não é um ato, é um hábito.”

Aristóteles

A estratégia e as táticas de teste sala limpa são fundamentalmente diferentes das abordagens de teste convencionais (Capítulos 17 até 20). Métodos convencionais derivam uma série de caso de testes para descobrir erros de projeto e codificação. O objetivo do teste sala limpa é validar requisitos de software demonstrando que uma amostragem estatística de casos de uso (Capítulo 5) foi executada de forma bem-sucedida.

21.4.1 Teste estatístico de uso

O usuário de um programa de computador raramente precisa entender os detalhes técnicos do projeto. O comportamento do programa visível ao usuário é controlado por entradas e eventos que muitas vezes são produzidos pelo usuário. Mas em sistemas complexos, o espectro possível de entradas e eventos (os casos de uso) pode ser extremamente amplo. Qual subconjunto de casos de uso verificará adequadamente o comportamento do programa? Essa é a primeira questão tratada no teste estatístico de uso.

O teste estatístico de uso “resume-se no teste do software da maneira que os usuários pretendem usá-lo” [Lin94b]. Para tanto, as equipes de teste sala limpa (também chamadas de *equipes de certificação*) devem determinar a distribuição de probabilidade de uso para o software. A especificação (caixa-preta) para cada incremento do software é analisada para determinar uma série de estímulos (entradas ou eventos) que fazem o software mudar seu comportamento. Com base em entrevistas com usuários em potencial, na criação de cenários de uso e em uma compreensão geral do domínio de aplicação, é atribuída a cada estímulo uma probabilidade de uso.

São gerados casos de testes para cada conjunto de estímulos⁴ de acordo com a distribuição de probabilidade de uso. Para ilustrar, considere o sistema *CasaSegura* já discutido neste



Mesmo que você decida ser contra a abordagem sala limpa, vale a pena considerar o teste estatístico de uso como parte integral de sua estratégia de teste.

4 Podem ser usadas ferramentas automáticas para conseguir isso. Para mais informações, veja [Dye92].

livro. A engenharia de software sala limpa está sendo usada para desenvolver um incremento de software que controle a interação do usuário com o teclado do sistema de segurança. Para esse incremento identificaram-se cinco estímulos. A análise indica a distribuição porcentual de probabilidade de cada estímulo. Para facilitar a seleção de casos de testes, essas probabilidades são mapeadas em intervalos numerados de 1 a 99 [Lin94] e ilustradas na tabela a seguir:

Estímulo ao programa	Probabilidade	Intervalo
Armar/Desarmar (AD)	50%	1–49
Definir zona (ZS)	15%	50–63
Consultar (Q)	15%	64–78
Testar (T)	15%	79–94
Alarme de pânico	5%	95–99

Para gerar uma sequência de casos de testes de uso que esteja de acordo com a distribuição de probabilidade de uso, geram-se números aleatórios entre 1 e 99. Cada número aleatório corresponde a um intervalo na distribuição de probabilidades mencionada. Portanto, a sequência de casos de testes de uso é definida aleatoriamente, mas corresponde a uma probabilidade apropriada de ocorrência de estímulo. Por exemplo, suponha que sejam geradas as seguintes sequências de números aleatórios:

13-94-22-24-45-56

81-19-31-69-45-9

38-21-52-84-86-4

Selecionando os estímulos apropriados com base no intervalo de distribuição mostrado na tabela, geram-se os seguintes casos de uso:

AD-T-AD-AD-AD-ZS

T-AD-AD-AD-Q-AD-AD

AD-AD-ZS-T-T-AD

A equipe de teste executa esses casos de uso e verifica o comportamento do software em relação à especificação para o sistema. São registrados os tempos para os testes de maneira que podem ser determinados os intervalos de tempo. Usando os intervalos de tempo, a equipe de certificação pode calcular o tempo médio para falhas (*mean-time-to-failure*, MTTF). Se for executada uma longa sequência de testes sem falha, o MTTF é baixo e a confiabilidade do software pode ser considerada alta.

21.4.2 Certificação

As técnicas de verificação e teste discutidas anteriormente neste capítulo levam a componentes de software (e incrementos completos) que podem ser certificados. No contexto da abordagem de engenharia de software sala limpa, a *certificação* implica que a confiabilidade (medida pelo MTTF) pode ser especificada para cada componente.

O impacto provável de componentes de software certificáveis vai muito além de um simples projeto sala limpa. Componentes de software reutilizáveis podem ser armazenados com seus cenários de uso, estímulos de programa e distribuições de probabilidade. Cada componente teria uma confiabilidade certificada sob o cenário de uso e regime de teste descrito. Essas informações são valiosas para aqueles que quiserem usar os componentes.

A abordagem de certificação envolve cinco passos [Woh94]: (1) devem ser criados cenários de uso, (2) é especificado um perfil de uso, (3) são gerados casos de testes com base no perfil, (4) são executados os testes, os dados de falhas são registrados e analisados, e (5) a confiabilidade

é calculada e certificada. Os passos 1 a 4 foram discutidos em uma seção anterior. A certificação para engenharia de software sala limpa requer a criação de três modelos [Poo93]:

Modelo de amostragem. O teste de software executa m casos de testes aleatórios e é certificado se não ocorrerem falhas ou se ocorrer um número especificado de falhas. O valor de m é derivado matematicamente para garantir que a confiabilidade desejada seja obtida.

Modelo de componente. Um sistema formado por n componentes deve ser certificado. O modelo de componente permite que o analista determine a probabilidade do componente i falhar antes de completar.

Modelo de certificação. A confiabilidade global do sistema é projetada e certificada.

Ao término do teste estatístico de uso, a equipe de certificação tem as informações necessárias para entregar um software com um MTTF certificado e computado usando cada um desses modelos. Se você estiver interessado em detalhes adicionais, veja [Cur86], [Mus87], ou [Poo93].

21.5 CONCEITOS DE MÉTODOS FORMAIS

A enciclopédia *The Encyclopedia of Software Engineering* [Mar01] define métodos formais da seguinte maneira:

Métodos formais usados no desenvolvimento de sistemas de computadores são técnicas baseadas matematicamente para descrever propriedades de sistemas. Esses métodos formais proporcionam uma estrutura dentro da qual as pessoas podem especificar, desenvolver e verificar sistemas de maneira sistemática em vez de *ad hoc*.

“Os métodos formais têm um tremendo potencial para melhorar a clareza e precisão das especificações de requisitos e para localizar erros importantes e sutis.”

Steve
Easterbrook
et al.

As propriedades desejadas de uma especificação formal — consistência, totalidade e ausência de ambiguidade — são os objetivos de todos os métodos de especificação. No entanto, a linguagem de especificação com base matemática, usada para métodos formais, resulta em uma possibilidade muito maior de obter essas propriedades. A sintaxe formal de uma linguagem de especificação (Seção 21.7) possibilita que os requisitos ou projeto sejam interpretados de uma única maneira, eliminando a ambiguidade que frequentemente ocorre quando uma linguagem natural (por exemplo, inglês) ou uma notação gráfica (por exemplo, UML) deve ser interpretada por um leitor. Os recursos descritivos da teoria dos conjuntos e notação lógica possibilitam definição clara dos requisitos. Para serem consistentes, os requisitos especificados em um ponto em uma especificação não podem ser contrariados em outro ponto. A consistência é obtida⁵ provando-se matematicamente que os fatos iniciais podem ser mapeados de maneira formal (usando regras de inferência) em declarações posteriores dentro da especificação.

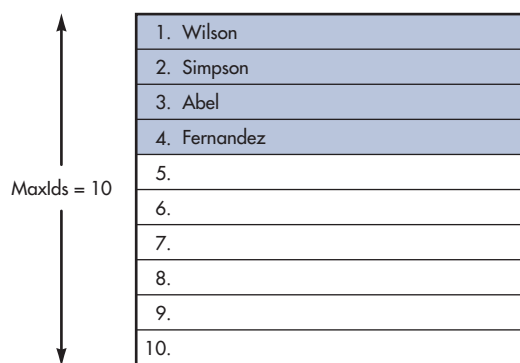
Para introduzirmos os conceitos básicos de métodos formais, vamos considerar alguns exemplos simples para ilustrar o uso da especificação matemática, sem aprofundar em detalhes da matemática.

Exemplo 1: Uma tabela de símbolos. Um programa é usado para manter uma tabela de símbolos. Emprega-se esse tipo de tabela frequentemente em diferentes tipos de aplicações. Ela consiste em uma coleção de itens sem qualquer duplicação. Um exemplo de uma típica tabela de símbolos está na Figura 21.7. Ela representa a tabela usada por um sistema operacional para manter os nomes dos usuários do sistema. Outros exemplos de tabelas incluem a coleção de nomes dos funcionários em um sistema de folha de pagamento, a coleção de nomes dos computadores em um sistema de comunicações em rede e a coleção de locais de destino em um sistema que gera as tabelas de horários de um sistema de transportes.

⁵ Na realidade, a totalidade é difícil de garantir, mesmo quando se utilizam métodos formais. Alguns aspectos de um sistema podem ser deixados indefinidos quando as especificações estão sendo criadas; outras características podem ser omitidas propositalmente para permitir que os projetistas tenham certa liberdade na escolha de abordagem de implementação; e, por fim, é impossível considerar cada cenário operacional em um sistema grande e complexo. Coisas podem ser omitidas por engano.

FIGURA 21.7

Uma tabela de símbolos



1. Wilson
2. Simpson
3. Abel
4. Fernandez
5.
6.
7.
8.
9.
10.

PONTO-CHAVE

Invariante de dados é um conjunto de condições que são verdadeiras ao longo de toda a execução do sistema que contém uma coleção de dados.



Outra maneira de entender o conceito de estado é dizer que os dados determinam o estado. Ou seja, você pode examinar os dados para ver em que estado o sistema está.

Suponha que a tabela apresentada neste exemplo seja formada apenas por nomes *MaxIds*. Essa afirmação, que impõe uma restrição na tabela, é o componente de uma condição conhecida como *invariante de dados*, uma condição verdadeira através de toda a execução do sistema que contém uma coleção de dados. A invariante de dados que se aplica à tabela de símbolos que acabamos de discutir tem duas componentes: (1) que a tabela terá apenas *MaxIds* e nada mais além disso e (2) que não haverá nomes duplicados na tabela. No caso do programa da tabela de símbolos, isso significa que não importa quando a tabela de símbolos será examinada durante a execução do sistema, ela sempre terá apenas *MaxIds* e nada mais, e não conterá duplicatas.

Outro conceito importante é o de *estado*. Muitas linguagens formais, como OCL (Seção 21.7.1), usam a noção de estado como discutido no Capítulo 7; isto é, um sistema pode estar em um dentre vários estados, cada um representando um modo de comportamento observável externamente. No entanto, uma definição diferente é usada na linguagem Z (Seção 21.7.2). Em Z (e linguagens relacionadas), o estado de um sistema é representado pelos dados do sistema armazenados (logo, Z sugere um número muito maior de estados, representando cada configuração possível dos dados). Usando a última definição no exemplo do programa da tabela de símbolos, o estado é a tabela de símbolos.

O conceito final é o de *operação*. Essa é uma ação que tem lugar em um sistema e lê ou escreve dados. Se o programa da tabela de símbolos está cuidando de adicionar ou remover nomes da tabela de símbolos, ele estará associado a duas operações: a operação *acrescentar()* um nome especificado à tabela de símbolos e a operação *remover()* da tabela um nome existente.⁶ Se o programa proporciona a facilidade de verificar se um nome específico está ou não contido na tabela, haverá uma operação que retornará alguma indicação sobre se o nome está ou não na tabela.

Três tipos de condições podem ser associadas a operações: invariantes, precondições e pós-condições. O *invariante* define o que está garantido que não mudará. Por exemplo, a tabela de símbolos tem uma invariante que diz que o número de elementos é sempre menor do que ou igual a *MaxIds*. Uma *precondição* define as circunstâncias nas quais uma operação em particular é válida. Por exemplo, a precondição para uma operação que acrescenta um nome a uma tabela de símbolos identificadores de funcionários é válida somente se o nome a ser adicionado não está contido na tabela e também se houver menos do que *MaxIds* identificadores de funcionários na tabela. A *pós-condição* de uma operação define o que é garantido que é verdadeiro após completar uma operação. Isso é definido por seu efeito sobre os dados. Para a operação *acrescentar()*, a pós-condição especificaria matematicamente que a tabela foi aumentada com o novo identificador.

⁶ Deve-se observar que a adição de um nome não pode ocorrer no estado *full* (cheio) e a exclusão de um nome é impossível no estado *empty* (vazio).

Exemplo 2: Um tratador de blocos. Uma das partes mais importantes de um sistema operacional simples é o subsistema que mantém os arquivos criados pelos usuários. Parte do subsistema de arquivos é o *tratador de blocos*. Os arquivos no sistema de armazenamento de arquivos são formados por blocos de armazenamento mantidos em um dispositivo de armazenamento. Durante a operação do computador, arquivos serão criados e excluídos, e isso requer a aquisição e liberação de blocos de armazenamento. Para enfrentar a tarefa, o subsistema de arquivamento deverá manter um reservatório de blocos não utilizados (livres) e manter controle dos blocos que estão correntemente em uso. Quando os blocos são liberados porque um arquivo foi excluído, eles normalmente são acrescentados a uma fila de blocos à espera de serem acrescentados ao reservatório de blocos não utilizados. Isso é apresentado na Figura 21.8. Nessa figura, há vários componentes: o reservatório de blocos não utilizados, os blocos que no momento formam os arquivos administrados pelo sistema operacional e aqueles blocos que estão esperando para ser acrescentados ao reservatório. Os blocos em espera são mantidos em fila, em que cada elemento da fila contém um conjunto de blocos de um arquivo excluído.



Técnicas de brainstorming podem funcionar bem quando você deve desenvolver uma invariante de dados para uma função razoavelmente complexa. Peça aos membros da equipe de software que escrevam os limites, as restrições e limitações para a função e então combinem e editem.

Para esse subsistema, o estado é a coleção de blocos livres, a coleção de blocos utilizados e a fila de blocos devolvidos. A invariante de dados, expressa em linguagem natural, é

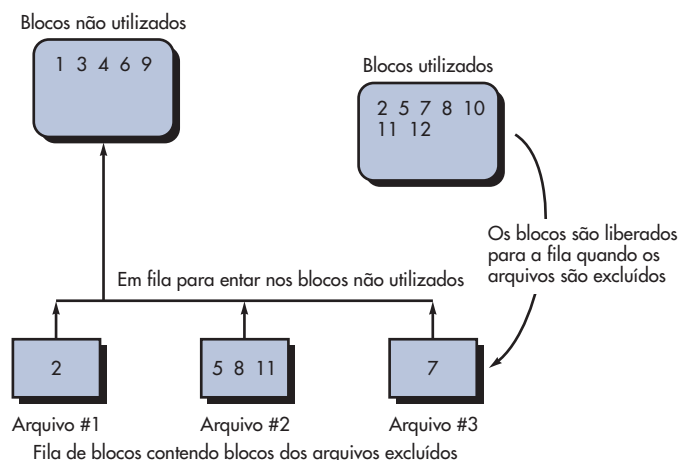
- Nenhum bloco será marcado como não usado e usado ao mesmo tempo.
- Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos usados correntemente.
- Nenhum elemento da fila terá o mesmo número de blocos.
- A coleção formada por blocos usados e blocos não usados será a coleção total dos blocos que formam os arquivos.
- A coleção de blocos não usados não terá número de blocos duplicados.
- A coleção de blocos usados não terá número de blocos duplicados.

Algumas das operações associadas a esses dados são: *acrescentar()* uma coleção de blocos ao fim da fila, *remover()* uma coleção de blocos usados da frente da fila e colocá-los em uma coleção de blocos não usados e *verificar()* se uma fila de blocos está vazia.

A pré-condição de *acrescentar()* é que os blocos a ser adicionados devem estar na coleção de blocos usados. A pós-condição é que a coleção de blocos agora se encontra no fim da fila. A pré-condição de *remover()* é que a fila deve ter pelo menos um item nela. A pós-condição é que os blocos devem ser adicionados à coleção de blocos não usados. A operação *verificar()* não tem pré-condição. Isso significa que a operação é sempre definida, independentemente do valor do estado. A pós-condição fornece o valor *true* (verdadeiro) se a fila estiver vazia e *false* (falso) caso contrário.

FIGURA 21.8

Tratador de blocos



Nos exemplos apresentados nesta seção, introduzi os conceitos fundamentais de especificação formal, mas sem ênfase à matemática necessária para tornar a especificação formal. Na Seção 21.6, considerarei como a notação matemática pode ser usada para especificar formalmente algum elemento de um sistema.

21.6 APLICANDO NOTAÇÃO MATEMÁTICA⁷ PARA ESPECIFICAÇÃO FORMAL

? Como posso representar estados e invariantes de dados usando um conjunto e operadores lógicos?

Para ilustrarmos o uso da notação matemática na especificação formal de um componente de software, examinamos novamente o exemplo de tratador de blocos apresentado na Seção 21.5. Para recordar, um componente importante do sistema operacional de um computador mantém arquivos que foram criados pelos usuários. O tratador de blocos mantém um reservatório de blocos não usados e manterá o controle de blocos que estão correntemente em uso. Quando os blocos são liberados de um arquivo excluído, eles são normalmente acrescentados à fila de blocos que estão à espera de ser adicionados ao reservatório de blocos não usados. Isso foi mostrado esquematicamente na Figura 21.8.

Um conjunto denominado *BLOCKS* será formado por todos os números de bloco. *AllBlocks* é um conjunto de blocos que estão entre 1 e *MaxBlocks*. O estado será modelado por dois conjuntos e uma sequência. Os dois conjuntos são *used* e *free*. Ambos contêm blocos — o conjunto *used* contém os blocos que estão correntemente em uso em arquivos, e o conjunto *free* contém blocos que estão disponíveis para novos arquivos. A sequência irá conter conjuntos de blocos que estão prontos para ser liberados dos arquivos que foram excluídos. O estado pode ser descrito como

used, free: $\mathbb{P} \text{ BLOCKS}$

BlockQueue: $\text{seq } \mathbb{P} \text{ BLOCKS}$

Isso é muito semelhante à declaração de variáveis de programa. Estamos dizendo que *used* e *free* serão conjuntos de blocos e que *BlockQueue* será uma sequência, em que cada elemento será um conjunto de blocos. O invariante de dados pode ser escrito como

$\text{used} \cap \text{free} = \emptyset \wedge$

$\text{used} \cup \text{free} = \text{AllBlocks} \wedge$

$\forall i: \text{dom } \text{BlockQueue} \bullet \text{BlockQueue } i \subseteq \text{used} \wedge$

$\forall i, j: \text{dom } \text{BlockQueue} \bullet i \neq j \Rightarrow \text{BlockQueue } i \cap \text{BlockQueue } j = \emptyset$

WebRef

Informações mais detalhadas sobre métodos formais podem ser encontradas em www.afm.sbu.ac.uk.

Os componentes matemáticos do invariante de dados correspondem a quatro dos componentes de linguagem natural destacados, descritos anteriormente. A primeira linha do invariante de dados informa que não haverá blocos comuns nas coleções de blocos usados e blocos livres. A segunda linha diz que a coleção de blocos usados e blocos livres sempre será igual à coleção total de blocos no sistema. A terceira linha indica que o *i*-ésimo elemento na fila de blocos sempre será um subconjunto dos blocos usados. A linha final diz que, para quaisquer dois elementos da fila de blocos que não sejam o mesmo bloco, não haverá blocos comuns nesses dois elementos. Os dois componentes finais de linguagem natural do invariante de dados são implementados em virtude do fato de que *used* e *free* são conjuntos e, portanto, não contêm duplicatas.

A primeira operação a ser definida é aquela que remove um elemento do início da fila de blocos. A precondition é que deve haver pelo menos um item na fila:

$\#\text{BlockQueue} > 0,$

⁷ Escrevi essa seção baseando-me na hipótese de que você está familiarizado com a notação matemática associada a conjuntos e sequências e a notação lógica usada no cálculo de predicado. Se você precisar de uma revisão, há um breve resumo apresentado como recurso suplementar no site da 7a. edição. Para informações mais detalhadas, veja [Jec06] ou [Pot04].

A pós-condição é que o início da fila deve ser removido e colocado na coleção de blocos livres e a fila deve ser ajustada para mostrar a remoção:

$$\begin{aligned} used' &= used \setminus head\ BlockQueue \wedge \\ free' &= free \cup head\ BlockQueue \wedge \\ BlockQueue' &= tail\ BlockQueue \end{aligned}$$

Uma convenção usada em muitos métodos formais é que o valor de uma variável após uma operação recebe uma indicação por meio de um apóstrofo. Portanto, o primeiro componente da expressão anterior diz que os novos blocos usados (*used'*) será igual aos blocos usados antigos menos os blocos removidos. O segundo componente diz que os novos blocos livres (*free'*) serão os blocos livres antigos com o início da fila de blocos acrescentado a eles. O terceiro componente diz que a nova fila de blocos será igual à cauda do valor antigo da fila de blocos, isto é, todos os elementos na fila exceto o primeiro. Uma segunda operação adiciona uma coleção de blocos, *Ablocks*, à fila de blocos. A pré-condição é que *Ablocks* seja correntemente um conjunto de blocos usados:

$$Ablocks \subseteq used$$

A pós-condição é que o conjunto de blocos seja acrescentado ao fim da fila de blocos e que o conjunto de blocos usados e livres permaneça inalterado:

$$\begin{aligned} BlockQueue' &= BlockQueue \smallfrown \langle Ablocks \rangle \wedge \\ used' &= used \wedge \\ free' &= free \end{aligned}$$

Não há dúvida de que a especificação matemática da fila de blocos é consideravelmente mais rigorosa do que uma narrativa em linguagem natural ou um modelo gráfico. O rigor adicional requer esforço, mas os benefícios obtidos da melhor consistência e completude podem ser justificados para alguns domínios de aplicação.



21.7 LINGUAGENS DE ESPECIFICAÇÃO FORMAL

Uma linguagem de especificação formal em geral é composta por três componentes primários: (1) uma sintaxe que define a notação específica com a qual a especificação é representada, (2) semânticas para ajudar a definir um “universo de objetos” [Win90] que será usado para descrever o sistema, e (3) um conjunto de relações que definem as regras que indicam quais objetos satisfazem adequadamente à especificação.

O domínio sintático de uma linguagem de especificação formal muitas vezes é baseado em sintaxe derivada da notação padrão de teoria dos conjuntos e do cálculo de predicado. O *domínio semântico* de uma linguagem de especificação indica como a linguagem representa os requisitos do sistema.

É possível usar diferentes abstrações semânticas para descrever o mesmo sistema de distintas maneiras. Fizemos isso de modo menos formal nos Capítulos 6 e 7; informações, funções e comportamentos foram representados. Diferentes notações de modelagem podem ser usadas para representar o mesmo sistema. As semânticas de cada representação proporcionam visões complementares do sistema. Para ilustrar essa abordagem quando são usados métodos formais, suponha que se utilize uma linguagem de especificação formal para descrever o conjunto de eventos responsável por um estado particular em um sistema. Outra relação formal mostra todas as funções que ocorrem em determinado estado. A intersecção dessas duas relações proporciona a indicação dos eventos responsáveis por funções específicas.

Há em uso hoje uma variedade de linguagens de especificação formal. OCL [OMG03b], Z [ISO02], LARCH [Gut93] e VDM [Jon91] são linguagens de especificação formal representativas que possuem as características já mencionadas. Neste capítulo, uma breve discussão da OCL e Z é apresentada.

TABELA 21.1 RESUMO DA NOTAÇÃO OCL FUNDAMENTAL

<code>x.y</code>	Obtém a propriedade <code>y</code> do objeto <code>x</code> . Uma propriedade pode ser um atributo, o conjunto de objetos no fim de uma associação, o resultado da avaliação de uma operação ou outros itens dependendo do tipo de diagrama UML. Se <code>x</code> for um Set, <code>y</code> será aplicado a todo elemento de <code>x</code> ; os resultados são reunidos em um novo Set.
<code>c->f()</code>	Aplica a operação <code>f</code> interna de OCL à própria Collection <code>c</code> (ao contrário de cada um dos objetos em <code>c</code>). A seguir estão listados exemplos de operações nativas.
<code>and</code> , <code>or</code> , <code>=</code> , <code><></code>	Operações lógicas <code>and</code> , <code>or</code> , <code>equals</code> , <code>not-equals</code> .
<code>p implies q</code>	Verdadeiro se <code>q</code> for verdadeiro ou <code>p</code> for falso.

Exemplos de operações sobre Collection (incluindo Sets e Sequences)

<code>C->size()</code>	O número de elementos na Collection <code>c</code> .
<code>C->isEmpty()</code>	Verdadeiro se <code>c</code> não tiver elementos, falso caso contrário.
<code>c1->includesAll(c2)</code>	Verdadeiro se todo elemento de <code>c2</code> for encontrado em <code>c1</code> .
<code>c1->excludesAll(c2)</code>	Verdadeiro se nenhum elemento de <code>c2</code> for encontrado em <code>c1</code> .
<code>C->forAll(elem l boolexpr)</code>	Verdadeiro se <code>boolexpr</code> for verdadeira quando aplicada a todo elemento de <code>c</code> . Quando um elemento está sendo avaliado, ele é ligado ao elemento variável, que pode ser usado em <code>boolexpr</code> . Isso implementa a quantificação universal, discutida anteriormente.
<code>C->forAll(elem1, elem2 l boolexpr)</code>	O mesmo que a anterior, exceto que <code>boolexpr</code> é avaliada para todo possível par de elementos tomados de <code>c</code> , incluindo casos em que o par consiste no mesmo elemento.
<code>C->isUnique(elem l expr)</code>	Verdadeiro se <code>expr</code> é avaliada a um valor diferente quando aplicada a cada elemento de <code>c</code> .

Exemplos de operações em Sets

<code>s1->intersection(s2)</code>	O conjunto daqueles elementos encontrados em <code>s1</code> e também em <code>s2</code> .
<code>s1->union(s2)</code>	O conjunto daqueles elementos encontrados em <code>s1</code> ou <code>s2</code> .
<code>s1->excluding(x)</code>	O conjunto <code>s1</code> com o objeto <code>x</code> omitido.

Exemplo de operação específica para Sequences

<code>Seq->first()</code>	O objeto que é o primeiro elemento na sequência <code>seq</code> .
------------------------------	--

21.7.1 Object Constraint Language (OCL)⁸

Object Constraint Language (OCL) é uma notação formal desenvolvida de forma que os usuários da UML possam adicionar mais precisão às suas especificações. Todos os poderes da lógica e da matemática discreta estão disponíveis na linguagem. No entanto, os projetistas da OCL decidiram que apenas os caracteres ASCII (em vez da notação matemática convencional) deveriam ser usados em instruções OCL. Isso torna a linguagem mais amigável àqueles menos inclinados à matemática e mais facilmente processada pelo computador. Mas isso torna a OCL um pouco prolixa em certas ocasiões.

Para usar a OCL, começa-se com um ou mais diagramas UML — mais comumente diagramas de classe, estado ou atividade (Apêndice 1). São acrescentadas as expressões OCL e declaram-se fatos sobre elementos dos diagramas. Essas expressões são chamadas de *restrições* (*constraints*); qualquer implementação derivada do modelo deve assegurar que cada uma das restrições permaneça sempre verdadeira.

⁸ Essa seção teve a contribuição do professor Timothy Lethbridge, da Universidade de Ottawa, e foi apresentada aqui com sua permissão.

Como uma linguagem de programação orientada a objeto, uma expressão OCL envolve operadores operando sobre objetos. No entanto, o resultado de uma expressão completa deve ser sempre booleano, isto é, verdadeiro ou falso. Os objetos podem ser instâncias da classe **Collection** da OCL, da qual **Set** e **Sequence** são duas subclasses.

O objeto **self** é o elemento do diagrama UML em cujo contexto a expressão OCL está sendo avaliada. Outros objetos podem ser obtidos por meio de *navegação* usando o símbolo `·` (dot) do objeto **self**. Por exemplo:

- Se **self** é a classe **C**, com atributo **a**, **self.a** avalia para objeto armazenado em **a**.
- Se **C** tem uma associação um-para-muitos chamada de *assoc* com outra classe **D**, **self.assoc** avalia para um **Set** cujos elementos são do tipo **D**.
- Por fim (e um pouco mais sutilmente), se **D** tem atributo **b**, a expressão **self.assoc.b** avalia para o conjunto de todos os **bs** pertencentes a todos os **Ds**.

A OCL proporciona operações internas implementando operadores set e logic, especificação construtiva e matemática relacionada. Um pequeno exemplo é apresentado na Tabela 21.1.

Para ilustrar o uso da OCL na especificação, reexaminamos o exemplo do tratador de blocos, introduzido na Seção 21.5. O primeiro passo é desenvolver um modelo UML (Figura 21.9). Esse diagrama de classe especifica muitas relações entre os objetos envolvidos. No entanto, são acrescentadas expressões OCL para permitir que os implementadores do sistema saibam mais precisamente o que deve permanecer verdadeiro enquanto o sistema processa.

As expressões OCL que suplementam o diagrama de classes correspondem às seis partes do invariante discutido na Seção 21.5. No exemplo a seguir, o invariante é repetido em inglês e então é escrita a expressão OCL correspondente. É aconselhável inserir texto de linguagem natural com a lógica formal; isso ajuda a entender a lógica e auxilia os revisores a descobrir os erros, por exemplo, situações em que não há correspondência entre o inglês e a lógica.

1. Nenhum bloco será marcado como não usado e usado.

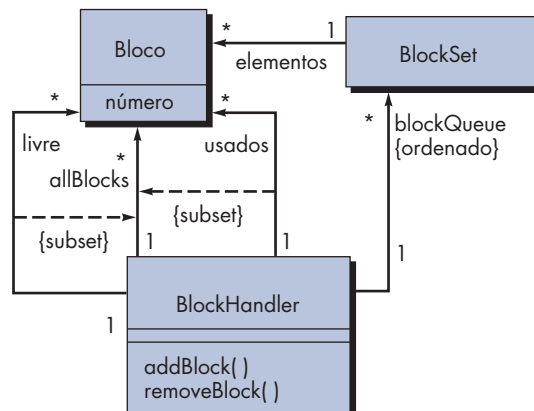
context BlockHandler inv:

(self.used -> intersection(self.free)) -> isEmpty()

Note que cada expressão começa com a palavra-chave **context**. Isso indica o elemento do diagrama UML que a expressão restringe. Como alternativa, você poderia colocar a restrição diretamente no diagrama UML, entre chaves {}. A palavra-chave **self** aqui se refere à instância de **BlockHandler**; no próximo item, como permite a OCL, vamos omitir o **self**.

FIGURA 21.9

Diagrama de classe para um tratador de blocos



2. *Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos usados correntemente.*

context BlockHandler inv:

blockQueue $\rightarrow$ forAll(aBlockSet | used $\rightarrow$ includesAll(aBlockSet))

3. *Nenhum elemento da fila terá o mesmo número de bloco.*

context BlockHandler inv:

blockQueue $\rightarrow$ forAll(blockSet1, blockSet2 |

blockSet1 $\leftrightarrow$ blockSet2 implies

blockSet1.elements.number $\rightarrow$ excludesAll(blockSet2.elements.number))

A expressão antes de **implies** é necessária para assegurar que ignoramos pares nos quais ambos os elementos são o mesmo bloco.

4. *A coleção de blocos usados e blocos que não são usados será a coleção total de blocos que compõem os arquivos.*

context BlockHandler inv:

allBlocks = used $\rightarrow$ union(free)

5. *A coleção de blocos não usados não terá números de bloco duplicados.*

context BlockHandler inv:

free $\rightarrow$ isUnique(aBlock | aBlock.number)

6. *A coleção de blocos usados não terá números de bloco duplicados.*

context BlockHandler inv:

used $\rightarrow$ isUnique(aBlock | aBlock.number)

A OCL também pode ser utilizada para especificar pré e pós-condições de operações. Por exemplo, a notação a seguir descreve operações que removem e acrescentam conjuntos de blocos à fila. Observe que a notação **x@pre** indica o objeto **x** da maneira como ele existe *antes* da operação; isso é o oposto da notação matemática discutida anteriormente, em que é o **x** *depois* da operação que é especialmente designado (como **x'**).

context BlockHandler::removeBlocks()

pre: blockQueue $\rightarrow$ size() > 0

post: used = used@pre-blockQueue@pre $\rightarrow$ first() and

free = free@pre $\rightarrow$ union(blockQueue@pre $\rightarrow$ first()) and

blockQueue = blockQueue@pre $\rightarrow$ excluding(blockQueue@pre $\rightarrow$ first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)

pre: used $\rightarrow$ includesAll(aBlockSet.elements)

post: (blockQueue.elements = blockQueue.elements@pre

$\rightarrow$ append (aBlockSet.elements) and

used = used@pre and

free = free@pre

A OCL é uma linguagem de modelagem, mas tem todos os atributos de uma linguagem formal. A OCL permite a expressão de várias restrições, pré e pós-condições, proteções e outras características relacionadas com objetos representados em vários modelos UML.

21.7.2 A linguagem de especificação Z

Z (que se pronuncia como “zed”) é uma linguagem de especificação amplamente usada na comunidade de métodos formais. A linguagem Z aplica conjuntos tipados, relações e funções no contexto da lógica de predicados de primeira ordem para criar *esquemas* — um meio de estruturar a especificação formal.

WebRef

Informações detalhadas sobre a linguagem Z podem ser encontradas em www.users.cs.york.ac.uk/~susan/abs/z.htm.

As especificações Z são organizadas como um conjunto de esquemas — uma estrutura de linguagem que introduz variáveis e especifica a relação entre essas variáveis. Um esquema é essencialmente a especificação formal análoga do componente de linguagem de programação. Esquemas são usados para estruturar uma especificação formal da mesma maneira que os componentes são utilizados para estruturar um sistema.

Um esquema descreve os dados armazenados que um sistema acessa e altera. No contexto de Z, isso é chamado de “estado”. O uso do termo *estado* em Z é ligeiramente diferente do empregado no restante do livro.⁹ Além disso, o esquema identifica as operações aplicadas para mudar o estado e as relações que ocorrem no sistema. A estrutura genérica de um esquema toma a forma:

Nome do esquema
declarações
invariante

em que declarações identificam as variáveis que formam o estado do sistema e a invariante impõe restrições sobre a maneira pela qual o estado pode evoluir. A Tabela 21.2 apresenta um resumo da notação da linguagem Z.

O exemplo a seguir de um esquema descreve o estado de um tratador de blocos e a invariante de dados:

BlockHandler
$used, free : \mathbb{P} BLOCKS$ $BlockQueue : seq \mathbb{P} BLOCKS$ $used \cap free = \emptyset \wedge$ $used \cup free = AllBlocks \wedge$ $\forall i: dom BlockQueue \bullet BlockQueue i \subseteq used \wedge$ $\forall i, j: dom BlockQueue \bullet i \neq j \Rightarrow BlockQueue i \cap BlockQueue j = \emptyset$

Conforme observamos, o esquema consiste em duas partes. A parte acima da linha central representa as variáveis do estado, enquanto a parte abaixo da linha central descreve a invariante de dados. Sempre que o esquema especificar operações que mudam o estado, ele é precedido pelo símbolo (Δ). O exemplo a seguir de um esquema descreve a operação que remove um elemento da fila de blocos:

RemoveBlocks
$\Delta BlockHandler$ $\#BlockQueue > 0,$ $used' = used \setminus head BlockQueue \wedge$ $free' = free \cup head BlockQueue \wedge$ $BlockQueue' = tail BlockQueue$

⁹ Lembre-se de que em outros capítulos, *estado* foi empregado para identificar um modo de comportamento observável externamente para um sistema.

TABELA 21.2 RESUMO DA NOTAÇÃO Z

A notação Z baseia-se na teoria de conjuntos típica e na lógica de primeira ordem. Z proporciona uma construção, chamada de esquema, para descrever o espaço de estado e operações de uma especificação. Um esquema agrupa declarações de variáveis com uma lista de predicados que restringem o valor possível de uma variável. Em Z, o esquema X é definido pela forma

X
declarações
predicados

Funções globais e constantes são definidas pela forma

declarações
predicados

A declaração fornece o tipo da função ou constante, enquanto o predicado dá seu valor. Nesta tabela há apenas um conjunto abreviado de símbolos Z.

Conjuntos:

$S : \mathbb{P} X$	S é declarada como um conjunto de X s.
$X \in S$	x é um membro de S .
$x \notin S$	x não é um membro de S .
$S \subseteq T$	S é um subconjunto de T : todo membro de S está também em T .
$S \cup T$	A união de S e T : contém todos os membros de S ou T ou ambos.
$S \cap T$	A intersecção de S e T : contém todos os membros de S e T .
$S \setminus T$	A diferença de S e T : contém todos os membros de S exceto aqueles que também estão em T .
$\emptyset$	Conjunto vazio: não contém membros.
$\{x\}$	Conjunto unitário: contém somente x .
$\mathbb{N}$	O conjunto de números naturais $0, 1, 2, \dots$
$S : \mathbb{F} X$	S é declarado como um conjunto finito de X s.
$\max(S)$	O máximo do conjunto não vazio de números S .

Funções:

$f: X \rightarrowtail Y$	f é declarada como uma injeção parcial de X para Y .
$\text{dom } f$	O domínio de f : o conjunto de valores x para os quais $f(x)$ é definida.
$\text{ran } f$	O intervalo de f : o conjunto de valores tomados por $f(x)$ quando x varia sobre o domínio de f .
$f \oplus \{x \mapsto y\}$	Uma função que concorda com f exceto que x é mapeado para y .
$\{x\} \triangleleft f$	Uma função como f , exceto que x é removido de seu domínio.

Lógica:

$P \wedge Q$	P e Q : é verdadeira se P e Q forem ambos verdadeiros.
$P \Rightarrow Q$	P implica Q : é verdadeira se Q for verdadeiro ou P for falso.
$\theta S' = \theta S$	Nenhum componente do esquema S muda em uma operação.

A inclusão do Δ *BlockHandler* resulta que todas as variáveis que formam o estado tornam-se disponíveis para o esquema *RemoveBlocks* e assegura que a invariante de dados será mantida antes e depois que a operação tiver sido executada.

A segunda operação, que acrescenta uma coleção de blocos ao fim da fila, é representada por

AddBlocks

Δ *BlockHandler*
 Ablocks? : BLOCKS

$Ablocks? \subseteq used$
 $BlockQueue' = BlockQueue \langle Ablocks? \rangle \wedge$
 $used' = used \wedge$
 $free' = free$

Por convenção em Z, uma variável de entrada que é lida, mas não forma parte do estado, é encerrada por um ponto de interrogação. Assim, Ablocks?, que age como um parâmetro de entrada, é encerrada por um ponto de interrogação.

FERRAMENTAS DO SOFTWARE



Métodos formais

Objetivo: o objetivo das ferramentas de métodos formais é ajudar uma equipe de software na especificação e verificação da correção.

Mecânica: a mecânica varia. Em geral, as ferramentas ajudam na especificação de uma prova automática da correção, usualmente definindo uma linguagem especializada para prova de teorema. Muitas ferramentas não são comercializadas e foram desenvolvidas para fins de pesquisa.

Ferramentas representativas:¹⁰

ACL2, desenvolvida na Universidade do Texas (www.cs.utexas.edu/users/moore/acl2/), é “tanto uma linguagem de programação na qual você pode modelar sistemas de computadores quanto uma ferramenta para ajudá-lo a provar propriedades daqueles modelos”.

EVES, desenvolvida pela ORA Canada (www.ora.on.ca/eves.html), implementa a linguagem Verdi para especificação formal e é um gerador automático de provas.

Uma extensa lista de mais de 90 ferramentas de métodos formais pode ser encontrada em <http://vl.fmnet.info/>.

21.8 RESUMO

A engenharia de software sala limpa é uma abordagem formal para o desenvolvimento de software que pode levá-lo a uma qualidade notavelmente alta. Ela usa a especificação de estrutura de caixa para análise e modelagem de projeto e dá ênfase à verificação da correção, em vez do teste, como mecanismo primário para localizar e remover erros. O teste de uso estatístico é aplicado para desenvolver as informações necessárias de taxa de falhas para certificar a confiabilidade do software fornecido.

A abordagem sala limpa começa com a análise e modelos de projeto que usam uma representação de estrutura de caixa. Uma “caixa” encapsula o sistema (ou algum aspecto do sistema) a um nível específico de abstração. Caixas-pretas são usadas para representar o comportamento de um sistema observável externamente. Caixas de estado encapsulam dados de estado e operações. Caixas-claras são usadas para modelar o projeto procedimental inferido pelos dados e operações de uma caixa de estado.

¹⁰ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

A verificação de correção é aplicada uma vez completo o projeto da estrutura de caixa. O projeto procedimental para um componente de software é particionado em uma série de subfunções. Para provar a correção das subfunções, definem-se condições de saída para cada subfunção e aplica-se uma série de subprovas. Se cada condição de saída é satisfeita, o projeto deve estar correto.

Uma vez completada a verificação da correção, inicia-se o teste estatístico de uso. Diferentemente do teste convencional, a engenharia de software sala limpa não enfatiza o teste de unidade ou de integração. Em vez disso, o software é testado definindo-se um conjunto de cenários de uso, determinando a probabilidade de uso para cada cenário e, por fim, definindo testes aleatórios que se sujeitam às probabilidades. Os registros de erros resultantes são combinados com os modelos de amostragem, componente e certificação para possibilitar a computação matemática da confiabilidade projetada para o componente de software.

Os métodos formais usam recursos descritivos da teoria dos conjuntos e notação lógica para possibilitar ao engenheiro de software criar uma definição clara dos fatos (requisitos). Os conceitos subjacentes que governam os métodos formais são: (1) a invariante de dados, uma condição verdadeira através de toda a execução do sistema que contém uma coleção de dados; (2) o estado, uma representação do modo de comportamento do sistema observável externamente ou (em Z e linguagens relacionadas) os dados armazenados que um sistema acessa e altera; e (3) a operação, uma ação que tem lugar em um sistema e lê ou escreve dados para um estado. A operação está associada a duas condições: uma pré e pós-condição.

Será que a engenharia de software sala limpa ou os métodos formais serão algum dia usados amplamente? A resposta é “provavelmente não”. Eles são mais difíceis de aprender do que os métodos de engenharia de software convencionais e representam um “choque cultural” significativo para alguns profissionais. Mas na próxima vez em que você ouvir alguém se lamentando, “Por que não podemos escrever esse software certo já na primeira vez?”, saberá que há técnicas que podem ajudá-lo a conseguir exatamente isso.

PROBLEMAS E PONTOS A PONDERAR

- 21.1.** Se você tivesse de escolher um aspecto da engenharia de software sala limpa que a torna radicalmente diferente das abordagens de engenharia de software convencionais orientadas a objeto, qual seria ele?
- 21.2.** Como um modelo de processo incremental e certificação funcionam em conjunto para produzir software de alta qualidade?
- 21.3.** Por meio da especificação de estrutura de caixa, desenvolva modelos de análise e projeto “primeira-passada” para o sistema *CasaSegura*.
- 21.4.** Um algoritmo de ordenação da bolha (bubble-sort) é definido da seguinte maneira:

```

procedure bubblesort;
var i, t, integer;
begin
  repeat until = 5 a[1]
    t := a[1];
    for j := 2 to n do
      if a[j-1] > a[j] then begin
        t := a[j-1];
        a[j-1] := a[j];
        a[j] := t;
      end
    endrep
  end

```

Particione o projeto em subfunções e defina um conjunto de condições que lhe possibilitariam provar que o algoritmo está correto.

21.5. Documente uma prova de verificação de correção para o bubble sort discutido no Problema 21.4.

21.6. Selecione um programa que você use regularmente (por exemplo, um programa de e-mail, um processador de texto, um programa de planilha). Crie um conjunto de cenários de uso para o programa. Defina a probabilidade do uso de cada cenário e depois desenvolva uma tabela de distribuição de probabilidades e estímulos de programa similar à apresentada na Seção 21.4.1.

21.7. Para a tabela de distribuição de estímulos e probabilidades de programa desenvolvida no Problema 21.6, use um gerador de números aleatórios para desenvolver um conjunto de caso de testes para um teste estatístico de uso.

21.8. Descreva o objetivo da certificação no contexto de engenharia de software sala limpa.

21.9. Você foi designado para uma equipe que está desenvolvendo software para um fax modem. A sua tarefa é desenvolver a parte da “lista telefônica” da aplicação. A função lista telefônica permite armazenar até *MaxNomes* de pessoas associadas ao nome da empresa, o número do fax e outras informações relacionadas. Usando linguagem natural, defina

- a. a invariante de dados.
- b. o estado.
- c. as operações possíveis.

21.10. Você foi designado para uma equipe de software que está desenvolvendo um programa chamado MemoryDoubler, que fornece para o PC uma memória aparentemente maior do que a memória física. Isso é conseguido identificando, coletando e reatribuindo blocos de memória que foram atribuídos a uma aplicação existente, mas que não estão sendo usados. Os blocos não usados são reatribuídos a aplicações que requerem memória adicional. Adotando as hipóteses apropriadas e usando linguagem natural, defina

- a. a invariante de dados.
- b. o estado.
- c. as operações possíveis.

21.11. Usando a notação OCL ou Z apresentadas na Tabela 21.1 ou 21.2, selecione alguma parte do sistema de segurança *CasaSegura* descrita anteriormente neste livro e tente especificá-la com OCL ou Z.

21.12. Usando uma ou mais das fontes de informações citadas nas referências deste capítulo ou em *Leituras e fontes de informação complementares*, desenvolva uma apresentação de meia hora sobre a sintaxe e semânticas básicas de uma linguagem de especificação formal que não seja OCL ou Z.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Nos anos recentes publicaram-se relativamente poucos livros sobre técnicas avançadas de especificação e verificação. No entanto, vale considerar algumas novas adições à literatura. Um livro editado por Gabbar (*Modern Formal Methods and Applications*, Springer, 2006) apresenta fundamentos, novos desenvolvimentos e aplicações avançadas. Jackson (*Software Abstractions*, The MIT Press, 2006) traz todos os fundamentos básicos e uma abordagem que ele chama de “métodos formais leves”. Monin e Hinchey (*Understanding Formal Methods*, Springer, 2003) proporcionam uma excelente introdução ao assunto. Butler e outros editores (*Integrated Formal Methods*, Springer, 2002) apresentam uma variedade de trabalhos sobre tópicos de métodos formais.

Além dos livros citados neste capítulo, Prowell e seus colegas (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) fornecem um tratamento profundo de todos os aspectos importantes da abordagem sala limpa. Discussões úteis dos tópicos sala limpa foram editadas por Poore e Trammell (*Cleanroom Software Engineering: A Reader*, Blackwell Publishing, 1996). Becker e Whittaker (*Cleanroom Software Engineering Practices*, Idea Group Publishing, 1996) apresentam uma excelente visão geral para aqueles que não estão familiarizados com as práticas sala limpa.

O Cleanroom Pamphlet (Software Technology Support Center, Hill AF Base, April 1995) contém reimpressões de vários artigos importantes. The Data and Analysis Center for Software (DACS) (www.dacs.dtic.mil) fornece muitas publicações úteis, guias e outras fontes de informação sobre engenharia de software sala limpa.

A verificação de projeto através da prova de correção está no coração da abordagem sala limpa. Livros de Cupillari (*The Nuts and Bolts of Proofs*, 3d ed., Academic Press, 2005), Solow (*How to Read and Do Proofs*, 4th ed., Wiley, 2004), Eccles (*An Introduction to Mathematical Reasoning*, Cambridge University Press, 1998) fornecem excelentes introduções aos fundamentos básicos matemáticos. Stavely (*Toward Zero-Defect Software*, Addison-Wesley, 1998), Baber (*Error-Free Software*, Wiley, 1991) e Schulmeyer (*Zero Defect Software*, McGraw-Hill, 1990) discutem prova de correção em detalhe considerável.

No domínio dos métodos formais, livros de Casey (*A Programming Approach to Formal Methods*, McGraw-Hill, 2000), Hinchey e Bowan (*Industrial Strength Formal Methods*, Springer-Verlag, 1999), Hussmann (*Formal Foundations for Software Engineering Methods*, Springer-Verlag, 1997) e Sheppard (*An Introduction to Formal Specification with Z and VDM*, McGraw-Hill, 1995) proporcionam diretrizes úteis. Além disso, livros específicos sobre linguagem como Warmer and Kleppe (*Object Constraint Language*, Addison-Wesley, 1998), Jacky (*The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997), Harry (*Formal Methods Fact File: VDM and Z*, Wiley, 1997) e Cooper e Barden (*Z in Practice*, Prentice-Hall, 1995) fornecem informações úteis para métodos formais, bem como uma variedade de linguagens de modelagem.

Uma ampla variedade de fontes de informações sobre engenharia de software sala limpa e métodos formais está disponível na Internet. Uma lista atualizada das referências relevantes na Web sobre modelagem formal e verificação pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

auditoria de configuração	526
controle de alterações	524
controle de versão	524
gestão de conteúdo	530
identificação	522
itens de configuração de software (SCIs) ..	518
itens de configuração	518
objetos de configuração de WebApp	529
processo SCM	521
referencial	516
relatório de status	527
repositório	519
WebApps	528

Mudanças são inevitáveis quando o software é construído. E as mudanças aumentam o nível de confusão entre os membros de uma equipe de software que estão trabalhando em um projeto. A confusão surge quando as mudanças não são analisadas antes de ser feitas, não são registradas antes de ser implementadas, não são relatadas àqueles que precisam saber, ou não são controladas de uma maneira que melhore a qualidade e reduza os erros. Babich [Bab86] discute isso quando afirma:

A arte de coordenar desenvolvimento de software para minimizar a... Confusão é chamada de gestão de configuração. A gestão de configuração é a arte de identificar, organizar e controlar modificações no software que está sendo criado por uma equipe de programação. O objetivo é maximizar a produtividade minimizando os erros.

A gestão de configuração de software (software configuration management – SCM) é uma atividade do tipo “guarda-chuva”, aplicada através de toda a gestão de qualidade. Como as mudanças podem ocorrer em qualquer instante, as atividades SCM são desenvolvidas para (1) identificar a alteração, (2) controlar a alteração, (3) assegurar que a alteração esteja sendo implementada corretamente e (4) relatar as alterações a outros interessados.

É importante fazer uma clara distinção entre suporte de software e gestão de configuração de software. Suporte é um conjunto de atividades de engenharia que ocorrem depois que o software foi fornecido ao cliente e posto em operação. Gestão de configuração é um conjunto de atividades de rastreamento e controle iniciadas quando um projeto de engenharia de software começa e termina apenas quando o software sai de operação.

Um objetivo primário da engenharia de software é incrementar a facilidade com que as alterações podem ser acomodadas e reduzir o esforço necessário quando as alterações

PANORAMA

O que é? Ao se criar software, acontecem mudanças. E por isso, precisamos gerenciá-las eficazmente. A gestão de configuração de software – (SCM), também chamada de gestão de alterações, é um conjunto de atividades destinadas a gerenciar as alterações identificando os artefatos que precisam ser alterados, estabelecendo relações entre eles, definindo mecanismos para gerenciar diferentes versões desses artefatos, controlando as alterações impostas e auditando e relatando as alterações feitas.

Quem realiza? Qualquer um que esteja envolvido na gestão de qualidade está envolvido com a gestão de alterações até certo ponto, mas muitas vezes criam-se posições especializadas de suporte para controlar o processo SCM.

Por que é importante? Se você não controlar as alterações, elas controlarão você. E isso nunca é bom. É muito fácil acontecer de uma sequência de alterações não controladas transformar um bom software em um caos. Como consequência, a qualidade do software é prejudicada e a entrega atrasa. Por essa razão, a gestão de alterações é parte essencial da gestão da qualidade.

Quais são as etapas envolvidas? Como muitos artefatos são produzidos quando o software é criado, cada um deles deve ser identificado de forma única. Feito isso, podem ser estabelecidos mecanismos para controle de versão e alteração. Para assegurar que a qualidade seja mantida quando são feitas alterações, o processo é auditado; e para assegurar que aqueles que precisam saber sobre as alterações sejam informados, são gerados relatórios.

Qual é o artefato? Um plano de gestão de configuração de software define a estratégia de projeto para a gestão das alterações. Além disso, quando é invocada a SCM formal, o processo de controle de alterações produz requisições de alteração de software, relatórios e ordens de alteração de engenharia.

Como garantir que o trabalho foi realizado corretamente? Quando cada artefato pode ser levado em conta, rastreado e controlado; quando todas as alterações podem ser rastreadas e analisadas, quando todos aqueles que precisam saber sobre as alterações já foram informados — você fez tudo certo.

tiverem de ser feitas. Neste capítulo, discutiremos as atividades específicas que lhe permitem gerenciar a alteração.

22.1 GESTÃO DE CONFIGURAÇÃO DE SOFTWARE

O resultado do processo de software são informações que podem ser divididas em três categorias principais: (1) programas de computador (tanto na forma de código-fonte quanto na forma executável), (2) produtos que descrevem os programas de computador (focado em vários interessados) e (3) dados ou conteúdo (contidos nos programas ou externos a ele). Os itens que compõem todas as informações produzidas como parte do processo de software são chamados coletivamente de *configuração de software*.

“Não há nada permanente, exceto as mudanças.”

Heráclito,
500 a.C.

À medida que avança o trabalho de engenharia de software, cria-se uma hierarquia de *itens de configuração de software* (*software configuration items* – SCIs) – um elemento de informação com nome, que pode ser tão pequeno quanto um simples diagrama UML ou tão grande quanto um documento de projeto completo. Se cada SCI simplesmente conduzir a outros SCIs, resultará em pouca confusão. Infelizmente, uma outra variável entra no processo – *alteração*. A alteração pode ocorrer a qualquer momento, por qualquer razão. De fato, a *Primeira Lei da Engenharia de Sistemas* [Ber80] diz: “Não importa onde você esteja no ciclo de vida do sistema, o sistema mudará e o desejo de alterá-lo persistirá através de todo o ciclo de vida”.

Qual é a origem dessas alterações? A resposta a essa pergunta é variada, assim como as próprias alterações. No entanto, há quatro fontes fundamentais de alterações:

- Novos negócios ou condições de mercado ditam mudanças nos requisitos do produto ou nas regras comerciais.
- Novas necessidades dos interessados demandam modificação dos dados produzidos pelos sistemas de informação, funcionalidade fornecida pelos produtos ou serviços fornecidos por um sistema baseado em computador.
- Reorganização ou crescimento/enxugamento causam alterações em prioridades de projeto ou estrutura de equipe de engenharia de software.
- Restrições orçamentárias ou de cronograma causam a redefinição do sistema ou produto.

A gestão de configuração de software é um conjunto de atividades que foram desenvolvidas para gerenciar alterações através de todo o ciclo de vida de um software. A SCM pode ser vista como uma atividade de garantia de qualidade do software aplicada através de todo o processo do software. Nas seções a seguir, descrevem-se as principais tarefas da SCM e conceitos importantes que podem ajudá-lo a gerenciar as alterações.

22.1.1 Um cenário SCM¹

Um típico cenário operacional CM envolve um gerente de projeto encarregado de um grupo de software, um gerente de configuração encarregado dos procedimentos e políticas CM, os engenheiros de software responsáveis pelo desenvolvimento e manutenção do artefato e o cliente que usa o produto. No cenário, suponha que o produto seja um item pequeno envolvendo aproximadamente 15 mil linhas de código que está sendo desenvolvido por uma equipe de seis pessoas. (Note que são possíveis outros cenários com equipes menores ou maiores, mas, essencialmente, há problemas genéricos que cada um desses projetos enfrenta em relação à CM.)

No nível operacional, o cenário envolve vários papéis e tarefas. Para o gerente de projeto, o objetivo é garantir que o produto seja desenvolvido em certo prazo. O gerente monitora o progresso do desenvolvimento e reconhece e reage aos problemas. Isso é feito gerando e analisando relatórios sobre o estado do sistema de software e fazendo revisões no sistema.

? Qual a origem das alterações solicitadas para o software?

? Quais os objetivos e as atividades executadas pelas divisões envolvidas na gestão de alterações?

¹ Esta seção foi extraída de [Dar01]. A permissão especial para reproduzir “Gama de Funcionalidade no Sistema CM” por Susan Dart [Dar01], © 2001 pelo Carnegie Mellon University, foi concedida pelo Software Engineering Institute.

As metas do gerente de configuração são garantir que sejam seguidos os procedimentos e políticas para criar, alterar e testar o código, bem como tornar acessíveis as informações sobre o projeto. Para implementar técnicas para manter controle sobre as mudanças de código, esse gerente introduz mecanismos para fazer solicitações oficiais de alterações, para avaliá-las (através de um Grupo de Controle de Alterações que é responsável pela aprovação das alterações do sistema) e para autorizar as alterações. O gerente cria e distribui listas de tarefas para os engenheiros e basicamente cria o contexto do projeto. Além disso, o gerente coleta dados estatísticos sobre os componentes do sistema de software, como, por exemplo, informações determinando que componentes do sistema são problemáticos.

Para os engenheiros de software, o objetivo é trabalhar eficazmente. Isso significa que os engenheiros não interferem uns com os outros de forma desnecessária na criação e teste do código e na produção de artefatos de suporte. Mas, ao mesmo tempo, eles tentam se comunicar e coordenar eficientemente. Os engenheiros usam ferramentas que ajudam a criar artefatos consistentes. Eles se comunicam e se coordenam notificando uns aos outros sobre as tarefas necessárias e as tarefas completadas. As alterações são propagadas por meio do trabalho dos outros mesclando arquivos. Existem mecanismos que asseguram que, para componentes submetidos a alterações simultâneas, há uma maneira de resolver conflitos e mesclar alterações. É mantido um histórico da evolução de todos os componentes do sistema juntamente com um registro (log) com as razões para as alterações e um registro do que realmente foi alterado. Os engenheiros têm seu próprio espaço de trabalho para criar, alterar, testar e integrar o código. Em certo ponto, o código é transformado em referencial com base no qual o desenvolvimento continua e por meio do qual são criadas variações para outras máquinas.

O cliente usa o produto. Como o produto está sob o controle da Gestão de Configuração (CM), o cliente segue os procedimentos formais para solicitar alterações e para indicar *bugs* no produto.

No caso ideal, um sistema de CM usado nesse cenário deveria suportar todos esses papéis e tarefas; isto é, os papéis determinam a funcionalidade requerida para um sistema de CM. O gerente de projeto vê a CM como um mecanismo de auditoria; o gerente de configuração a vê como um mecanismo de controle, rastreamento e criador de políticas; o engenheiro de software a vê como um mecanismo de alteração, criação e controle de acesso; e o cliente a vê como um mecanismo de garantia de qualidade.

22.1.2 Elementos de um sistema de gestão de configuração

Em sua publicação sobre gestão de configuração de software, Susan Dart [Dar01] identifica quatro importantes elementos que devem existir quando é desenvolvido um sistema de gestão de configuração:

- *Elementos de componente* — conjunto de ferramentas acopladas em um sistema de gestão de arquivos (por exemplo, um banco de dados) que possibilita acesso à gestão de cada item de configuração de software.
- *Elementos de processo* — coleção de ações e tarefas que definem uma abordagem eficaz da gestão de alterações (e atividades relacionadas) para todas as partes envolvidas na gestão, engenharia e uso do software.
- *Elementos de construção* — conjunto de ferramentas que automatizam a construção do software, assegurando que tenha sido montado o conjunto apropriado de componentes validados (isto é, a versão correta).
- *Elementos humanos* — conjunto de ferramentas e características de processo (abrangendo outros elementos de CM) usados pela equipe de software para implementar uma SCM eficaz.

Esses elementos (discutidos com mais detalhes em seções posteriores) não são mutuamente exclusivos. Por exemplo, elementos de componente funcionam em conjunto com elementos de cons-

PONTO-CHAVE

Deve haver um mecanismo para assegurar que alterações simultâneas ao mesmo componente sejam adequadamente rastreadas, gerenciadas e executadas.

trução à medida que evolui o processo do software. Elementos de processo guiam muitas atividades humanas relacionadas à SCM e podem, portanto, ser consideradas elementos humanos também.

22.1.3 Referenciais



Muitas alterações de software são justificadas, portanto não tem sentido reclamar delas. Em vez disso, esteja certo de ter os mecanismos prontos para cuidar delas.

Alteração é um fato normal no desenvolvimento de software. Clientes querem modificar requisitos. Desenvolvedores querem modificar a abordagem técnica. Gerentes querem modificar a estratégia do projeto. Por que todas essas modificações? A resposta é realmente muito simples.

À medida que o tempo passa, todas as partes envolvidas sabem mais (sobre o que eles precisam, qual será a melhor abordagem e como conseguir que seja feito e ainda ganhar dinheiro). Esse conhecimento adicional é a força motora que está por trás da maioria das alterações e leva à constatação de um fato que para muitos profissionais de engenharia de software é difícil de aceitar: *Muitas alterações são justificadas!*

Uma referência é um conceito de gestão de configuração de software que o ajuda a controlar alterações sem obstruir seriamente as alterações justificáveis. O IEEE (IEEE Std. No. 610.12-1990) define uma referência como:

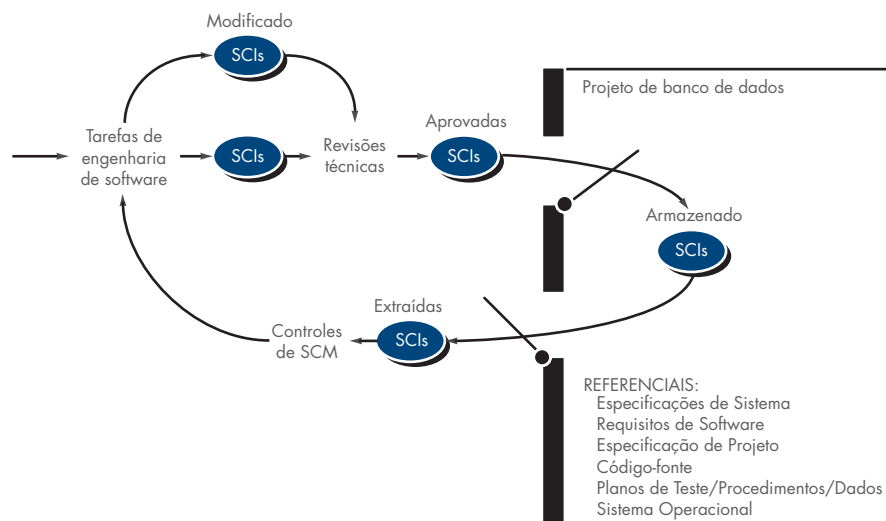
Uma especificação ou produto que tenha sido formalmente revisado e acordado, que depois serve de base para mais desenvolvimento, e pode ser alterado somente por meio de procedimentos formais de controle de alteração.

Para que um item de configuração de software se torne uma referência para o desenvolvimento, devem ser feitas alterações rápida e informalmente. No entanto, uma vez estabelecida uma referência, podem ser feitas alterações, mas deve ser aplicado um processo específico e formal para avaliar e verificar cada alteração.

No contexto de engenharia de software, uma referência é um marco no desenvolvimento de software. Uma referência é marcada pelo fornecimento de um ou mais itens de configuração de software que foram aprovados em consequência de uma revisão técnica (Capítulo 15). Por exemplo, os elementos de um modelo de projeto foram documentados e revisados. Erros foram encontrados e corrigidos. Uma vez que todas as partes do modelo foram revisadas, corrigidas e então aprovadas, o modelo do projeto torna-se uma referência. Outras alterações na arquitetura do programa (documentadas no modelo de projeto) podem ser feitas apenas depois que cada uma tenha sido avaliada e aprovada. Embora as referências possam ser definidas em qualquer nível de detalhe, as referências de software mais comuns estão na Figura 22.1.

FIGURA 22.1

SCIs que se tornaram referenciais e o banco de dados de projeto





Certifique-se de que o banco de dados de projeto seja mantido em uma localização centralizada e controlada.

A sequência de eventos que levam a uma referência também está ilustrada na Figura 22.1. Tarefas de engenharia de software produzem uma ou mais SCIs. Depois que as SCIs são revisadas e aprovadas, são colocadas em um *banco de dados de projeto* (também chamado de *biblioteca de projeto* ou *repositório de software* e discutidos na Seção 22.2). Quando um membro de uma equipe de engenharia de software quer fazer uma modificação em uma SCI que se tornou referencial, ela é copiada do banco de dados de projeto para o espaço de trabalho privado do engenheiro. Porém, essa SCI extraída só pode ser modificada se os controles de SCM (discutidos mais adiante neste capítulo) forem seguidos. As setas na Figura 22.1 ilustram o caminho de modificação para uma SCI referencial.

22.1.4 Itens de configuração de software

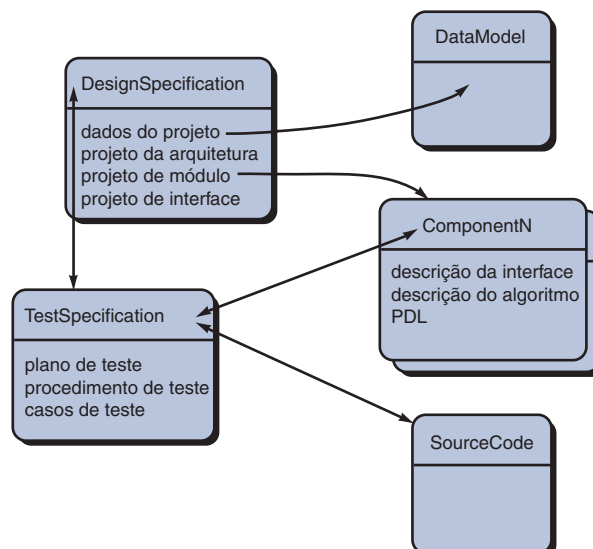
Já definimos um item de configuração de software como informação criada como parte do processo de engenharia de software. No caso extremo, uma SCI poderia ser considerada uma única seção de uma grande especificação ou um caso de teste em um grande conjunto de testes. Mais realisticamente, uma SCI é todo ou parte de um artefato (por exemplo, um documento, um conjunto inteiro de casos de teste ou um programa ou componente com nome).

Além das SCIs derivadas dos artefatos de software, muitas organizações de engenharia de software também colocam ferramentas de software sob o controle de configuração. Isto é, versões específicas de editores, compiladores, browsers e outras ferramentas automáticas são “congeladas” como parte da configuração do software. Como essas ferramentas foram usadas para produzir documentação, código-fonte e dados, elas devem estar disponíveis quando alterações forem feitas na configuração do software. Embora os problemas sejam raros, é possível que uma nova versão de uma ferramenta (por exemplo, um compilador) possa produzir resultados diferentes daqueles da versão original. Por essa razão, as ferramentas, assim como o software que elas ajudam a produzir, podem ser referenciadas como parte de um processo bem especificado de gestão de configuração.

Na realidade, as SCIs são organizadas para formar objetos de configuração que podem ser catalogados no banco de dados do projeto com um nome único. Um *objeto de configuração* tem um nome, atributos e é “conectado” a outros objetos por relações. De acordo com a Figura 22.2, os objetos de configuração, **DesignSpecification**, **DataModel**, **ComponentN**, **SourceCode** e **TestSpecification** são definidos separadamente. No entanto, cada um dos objetos está relacionado com os outros, como mostram as setas. Uma seta curva indica uma relação de composição. Isto é, **DataModel** e **ComponentN** são parte do objeto **DesignSpecification**. Uma seta

FIGURA 22.2

Objetos de configuração



reta bidirecional indica uma inter-relação. Se for feita uma alteração no objeto **SourceCode**, as inter-relações lhe permitem determinar que outros objetos (e SCIs) podem ser afetados.²

22.2 O REPOSITÓRIO SCM

Nos primórdios da engenharia de software, os itens de configuração eram mantidos na forma de documentos impressos (ou cartões perfurados!), colocados em pastas de arquivos ou naquelas pastas de três furos, e armazenados em armários de aço. Essa abordagem era problemática por várias razões: (1) encontrar um item de configuração quando necessário era muitas vezes difícil, (2) determinar quais itens foram alterados, quando e por quem, era em geral um desafio, (3) criar uma nova versão de um programa existente era um processo demorado e sujeito a erros e (4) descrever relações detalhadas ou complexas entre itens de configuração era praticamente impossível.

Atualmente, as SCIs são mantidas em um banco de dados de projeto ou repositório. O Dicionário Webster define a palavra *repository* (repositório) como “qualquer coisa ou pessoa considerada um centro de acumulação ou armazenagem”. Nos primeiros tempos da engenharia de software, o repositório era sem dúvida uma pessoa — o programador que tinha de se lembrar da localização de todas as informações relevantes a um projeto de software, aquele que tinha de se lembrar de todas as informações que nunca foram escritas e reconstruir informações perdidas. Infelizmente, o uso de uma pessoa como “centro de acumulação e armazenagem” (embora esteja de acordo com a definição *Webster’s Dictionary*) não funciona muito bem. Hoje, o repositório é uma “coisa” — um banco de dados que age como o centro de acumulação e de armazenagem de informações de engenharia de software. O papel da pessoa (o engenheiro de software) é interagir com o repositório usando ferramentas integradas com ele.

22.2.1 O papel do repositório

O repositório de SCM é um conjunto de mecanismos e estruturas de dados que permitem a uma equipe de software gerenciar alterações de maneira eficaz. Ele proporciona as funções óbvias de um sistema moderno de gestão de banco de dados garantindo a integridade dos dados, compartilhamento e integração. Além disso, o repositório de SCM proporciona um centralizador (*hub*) para a integração das ferramentas de software, está no centro do fluxo do processo de software e pode impor estrutura e formato uniformes para os artefatos.

Para tanto, o repositório é definido em termos de metamodelo. O *metamodelo* determina como as informações são armazenadas no repositório, como os dados podem ser acessados pelas ferramentas e visualizados pelos engenheiros de software, quão bem pode ser mantida a segurança e a integridade dos dados e com que facilidade o modelo existente pode ser estendido para satisfazer a novas necessidades.

22.2.2 Características gerais e conteúdo

As características e o conteúdo do repositório são mais bem compreendidas quando são observadas a partir de duas perspectivas: o que tem de ser armazenado no repositório e quais os serviços específicos que são fornecidos pelo repositório. Na Figura 22.3 está uma divisão detalhada dos tipos de representações, documentos e outros produtos que são armazenados no repositório.

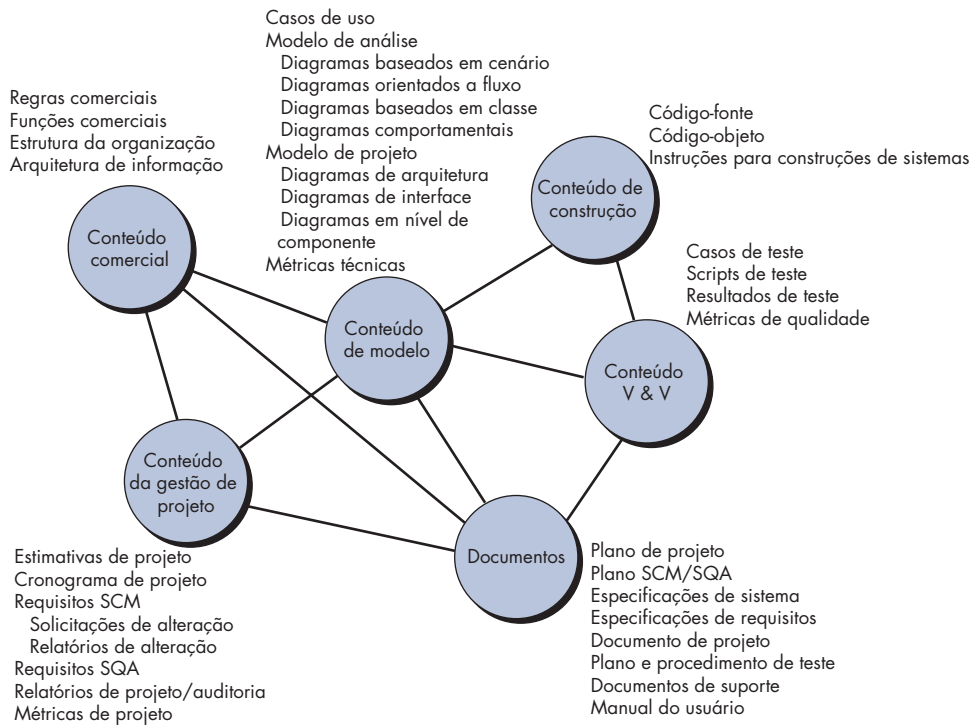
Um repositório robusto proporciona duas classes diferentes de serviços: (1) os mesmos tipos de serviços que podem ser esperados de qualquer sistema sofisticado de gerenciamento de banco de dados (2) serviços que são específicos ao ambiente de engenharia de software.

Um repositório que sirva a uma equipe de engenharia de software deve também ter as seguintes características (1) integrar com ou suportar diretamente as funções de gestão de processo,

WebRef

Um exemplo de um repositório disponível comercialmente pode ser obtido em www.oracle.com/technology/products/repository/index.html.

² Essas relações são definidas no banco de dados. A estrutura do banco de dados (repositório) é discutida em mais detalhes na Seção 22.2.

FIGURA 22.3**Conteúdo do repositório**

(2) suportar regras específicas que governam a função SCM e os dados mantidos no repositório, (3) proporcionar uma interface para outras ferramentas de engenharia de software, e (4) acomodar o armazenamento de objetos de dados sofisticados (por exemplo, texto, gráficos, vídeo, áudio).

22.2.3 Características SCM

Para suportar a SCM, o repositório deve ter um conjunto de ferramentas que proporcione suporte para as seguintes características:

Versões. À medida que um projeto avança, serão criadas muitas versões (Seção 22.3.2) dos artefatos individuais. O repositório deve ser capaz de salvar todas essas versões para possibilitar uma gestão eficaz das versões do produto e permitir aos desenvolvedores retroceder a versões anteriores durante o teste e depuração.

O repositório deve ser capaz de controlar uma grande variedade de tipos de objetos, incluindo texto, gráficos, bitmaps, documentos complexos e objetos especiais como definições de tela e relatório, arquivos de objeto, dados de testes e resultados. Um repositório desenvolvido rastreia versões de objetos com níveis arbitrários de granularidade; por exemplo, podem ser rastreados uma definição de dados especial ou um conjunto de módulos.

Acompanhamento de dependências e gestão de alterações. O repositório gerencia uma grande variedade de relações entre os elementos de dados armazenados nele. Isso inclui relações entre entidades e processos corporativos, entre as partes do projeto de uma aplicação, entre componentes de projeto e arquitetura de informações corporativas, entre elementos de projeto e outros artefatos, e assim por diante. Algumas dessas relações são meramente associações e outras são relações de dependências ou de obrigatoriedade.

PONTO-CHAVE

O repositório deve ser capaz de manter SCIs relacionadas com muitas versões diferentes do software. Mais importante ainda, deve proporcionar os mecanismos para montagem dessas SCIs em uma configuração específica de versão.

A habilidade em manter controle de todas essas relações é crucial para a integridade das informações armazenadas no repositório e para a geração de outros produtos baseados nele e é uma das contribuições mais importantes do conceito de repositório para o aperfeiçoamento do processo de software. Por exemplo, se um diagrama de classe UML é modificado, o repositório pode detectar se as classes relacionadas, as descrições de interface e os componentes de código também requerem modificações e podem chamar a atenção do desenvolvedor para as CSIs afetadas.

Controle de requisitos. Essa função especial depende da gestão de link e proporciona a habilidade para controlar todos os componentes de projeto e construção e outros produtos que resultam de uma especificação especial de requisitos (acompanhamento adiante). Além disso, ela proporciona a habilidade para identificar que requisitos geraram determinado produto (retroacompanhamento).

Gestão de configuração. O recurso de gestão de configuração mantém controle de uma série de configurações representando marcos de projeto específico ou versões de produção.

Pistas de auditoria. Uma pista de auditoria estabelece informações adicionais sobre quando, por que e por quem foram feitas as alterações. As informações sobre a origem das alterações podem ser colocadas como atributos de objetos específicos no repositório. Um mecanismo de disparo do repositório é útil para avisar o desenvolvedor ou a ferramenta que está sendo usada para iniciar a aquisição de informações de auditoria (como, por exemplo, a razão de uma alteração) sempre que um elemento de projeto for modificado.

22.3 O PROCESSO SCM

“Qualquer alteração, mesmo uma alteração para melhor, é acompanhada de contratempos e desconfortos.”

Arnold Bennett

 **Que questões o processo SCM está designado a responder?**

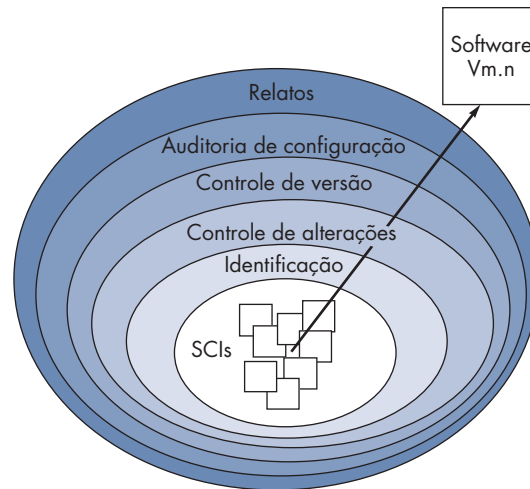
O processo de gestão de configuração de software define uma série de tarefas que têm quatro objetivos primários: (1) identificar todos os itens que coletivamente definem a configuração do software, (2) gerenciar alterações de um ou mais desses itens, (3) facilitar a construção de diferentes versões de uma aplicação e (4) assegurar que a qualidade do software seja mantida à medida que a configuração evolui com o tempo.

Um processo que atinja esses objetivos não precisa ser burocrático ou pesado, mas deve ser caracterizado de maneira que permita a equipe de software desenvolver respostas a uma série de questões complexas:

- Como uma equipe de software identifica os elementos discretos de uma configuração de software?
- Como uma organização lida com as várias versões de um programa (e sua documentação) de maneira que venha a permitir que a alteração seja acomodada eficientemente?
- Como uma organização controla alterações antes e depois que o software é entregue ao cliente?
- Quem tem a responsabilidade de aprovar e classificar as alterações solicitadas?
- Como podemos assegurar que as alterações foram feitas corretamente?
- Que mecanismo é usado para alertar outras pessoas sobre as alterações que são feitas?

Essas questões levam à definição de cinco tarefas SCM — identificação, controle de versão, controle de alteração, auditoria de configuração e relatos — ilustradas na Figura 22.4.

De acordo com a figura, as tarefas SCM podem ser vistas como camadas concêntricas. As SCIs fluem para fora através dessas camadas por toda a sua vida útil, tornando-se finalmente parte da configuração do software de uma ou mais versões da aplicação ou sistema. À medida que uma SCI se move através de uma camada, as ações deduzidas por cada tarefa SCM podem ser ou não aplicáveis. Por exemplo, quando uma nova SCI é criada, ela deve ser identificada. No entanto, se não forem solicitadas alterações para a SCI, a camada de controle de alteração não se aplica. A SCI é atribuída a uma versão específica do software (aqui entram em ação os mecanismos de controle de versão). É mantido um registro da SCI (seu nome, data de criação,

FIGURA 22.4**Camadas do processo SCM**

designação da versão etc.) para fins de auditoria da configuração e relato para aqueles que precisam ter conhecimento. Nas próximas seções, examinaremos cada uma dessas camadas de processo SCM em mais detalhes.

22.3.1 Identificação de objetos na configuração de software

Para controlar e gerenciar itens de configuração de software, cada um deles deverá ser nomeado separadamente e depois organizado usando uma abordagem orientada a objeto. Podem ser identificados dois tipos de objeto [Cho89]: objetos básicos e objetos agregados.³ O *objeto básico* é uma unidade de informação que se cria durante a análise, projeto, codificação ou teste. Por exemplo, o objeto básico pode ser uma seção de especificação de requisitos, parte de um modelo de projeto, código-fonte para um componente ou um conjunto de casos de teste usados para exercitar o código. O *objeto agregado* é uma coleção de objetos básicos e outros objetos agregados. Por exemplo, uma **DesignSpecification** é um objeto agregado. Conceitualmente, ela pode ser vista como uma lista nomeada (identificada) de ponteiros que especificam objetos agregados como, por exemplo, **ArchitecturalModel** e **DataModel**, e *objetos básicos* como **ComponentN** e **UMLClassDiagramN**.

Cada objeto tem um conjunto de características distintas que o identificam de forma única: um nome, uma descrição, uma lista de recursos e uma “realização”. O nome do objeto é uma sequência de caracteres que o identifica de forma não ambígua. A descrição do objeto é uma lista de itens de dados que identifica o tipo de SCI (por exemplo, elemento de modelo, programa, dados) representado pelo objeto, um identificador de projeto e informações sobre alteração e/ou versão. Recursos são “entidades fornecidas, processadas, referenciadas ou de qualquer outra forma requeridas pelo objeto” [Cho89]. Por exemplo, tipos de dados, funções específicas ou mesmo nomes de variáveis podem ser considerados recursos de objeto. A realização pode ser um ponteiro para a “unidade de texto”, para um objeto básico null ou para um objeto agregado.

A identificação do objeto de configuração pode também considerar as relações entre objetos nomeados. Por exemplo, usando a notação simples

```
Class diagram <part-of> requirements model;
Requirements model <part-of> requirements specification;
```

você pode criar uma hierarquia de SCIs.

PONTO-CHAVE

As inter-relações estabelecidas para objetos de configuração lhe permitem avaliar o impacto da alteração.

³ O conceito de objeto agregado [Gus89] tem sido proposto como mecanismo para representar uma versão completa de uma configuração de software.



Mesmo que o banco de dados do projeto tenha habilidade para estabelecer essas relações, elas demoram para se estabelecer e são difíceis de manter atualizadas. Embora muito úteis para análise de impacto, não são essenciais para a gestão geral das alterações.

Em muitos casos, objetos são inter-relacionados através de ramificações da hierarquia do objeto. Essas relações que cruzam a estrutura podem ser representadas da seguinte maneira:

```
DataModel <interrelated> DataFlowModel
DataModel <interrelated> TestCaseClassM
```

No primeiro caso, a inter-relação é entre um objeto composto, enquanto a segunda relação é entre um objeto agregado (**DataModel**) e um objeto básico (**TestCaseClassM**).

O esquema de identificação para objetos de software deve reconhecer que objetos evoluem através do processo de software. Antes que um objeto seja um referencial, ele pode mudar muitas vezes, e mesmo após um referencial ter sido estabelecido, as mudanças podem ser muito frequentes.

22.3.2 Controle de versão

O controle de versão combina procedimentos e ferramentas para gerenciar diferentes versões dos objetos de configuração criados durante o processo de software. Um sistema de controle de versão implementa ou está diretamente integrado com quatro recursos principais: (1) uma base de dados de projeto (repositório) que armazena todos os objetos de configuração relevantes, (2) um recurso de *gestão de versão* que armazena todas as versões de um objeto de configuração (ou permite que qualquer versão seja construída usando diferenças das versões anteriores), (3) uma facilidade de construir que permite coletar todos os objetos de configuração relevantes e construir uma versão específica do software. Além disso, os sistemas de controle de versão e controle de alteração muitas vezes implementam um recurso chamado acompanhamento de tópicos (também conhecido como acompanhamento de *bug*), que permite à equipe de software registrar e acompanhar o status de todos os problemas pendentes associados com cada objeto de configuração.

Alguns sistemas de controle de versão criam um conjunto de modificações — uma coleção de todas as alterações (em relação a alguma configuração referencial) que são necessárias para criar uma versão específica do software. Dart [Dar91] observa que um conjunto de modificações “captura todas as alterações a todos os arquivos na configuração juntamente com a razão para aquelas alterações e os detalhes de quem as fez e quando”.

Alguns conjuntos de modificações que receberam denominação podem ser identificados para uma aplicação ou sistema. Isso permite construir uma versão do software especificando os conjuntos de modificações (pelo nome) que devem ser aplicados à configuração referencial. Para isso, aplica-se uma abordagem de *modelagem de sistema*. O modelo de sistema contém: (1) um gabarito que inclui hierarquia de componentes e uma “ordem de construção” para os componentes descrevendo como o sistema deve ser construído, (2) regras de construção e (3) regras de verificação.⁴

Durante as últimas décadas foram propostas muitas abordagens automáticas diferentes para o controle de versão. A diferença primária entre as abordagens é a sofisticação dos atributos usados para construir versões específicas e variantes de um sistema e os mecanismos do processo de construção.

Mesmo com essas limitações, o CVS “é um sistema de controle de versão predominante de código aberto transparente à rede [que] é útil para qualquer um, desde desenvolvedores individuais até grandes equipes distribuídas” [CVS07]. Sua estrutura cliente-servidor permite aos usuários acessar arquivos via conexões na Internet, e sua filosofia de código aberto o torna disponível às plataformas mais populares.

O CVS está disponível sem custo para ambientes Windows, Mac OS, LINUX e UNIX. Ver [CVS07] para mais detalhes.

⁴ É possível também consultar o modelo do sistema para avaliar como uma alteração em um componente afeta outros componentes.

FERRAMENTAS DO SOFTWARE



O sistema de versões concorrentes (concurrent versions system – CVS)

O uso de ferramentas para deter o controle de versão é essencial para uma gestão eficaz das alterações. O sistema de versões concorrentes (CVS) é uma ferramenta largamente utilizada para controle de versão. Projetada originalmente para código-fonte, mas útil para qualquer arquivo baseado em texto, o sistema CVS (1) estabelece um repositório simples, (2) mantém todas as versões de um arquivo sob um único nome de arquivo, armazenando apenas as diferenças entre versões progressivas do arquivo original e (3) protege contra

alterações simultâneas de um arquivo, estabelecendo diferentes diretórios para cada desenvolvedor, isolando assim uns dos outros. O CVS mescla as alterações quando cada desenvolvedor completa seu trabalho.

É importante notar que o CVS não é um sistema de construção; ele não constrói uma versão específica do software. Outras ferramentas (por exemplo, *Makefile*) devem ser integradas ao CVS para conseguir isso. O CVS não implementa um processo de controle de alteração (por exemplo, solicitações de alterações, relatos de alterações, acompanhamento de bugs).

22.3.3 Controle de alterações

A realidade do controle de alterações em um moderno contexto de engenharia de software foi resumida elegantemente por James Bach [Bac98]:

O controle de alterações é vital. Mas as forças que o tornam necessário também o tornam inconveniente. Temos medo das alterações porque uma pequena perturbação no código pode criar uma enorme falha no produto. Mas elas podem também reparar uma grande falha ou habilitar novos e maravilhosos recursos. Temos medo das alterações porque um único desenvolvedor irresponsável poderia afundar o projeto todo; embora ideias brilhantes possam surgir nas mentes desses brincalhões, um controle de processo de alterações pesado poderia efetivamente desencorajá-los no seu trabalho criativo.

Bach reconhece que temos aqui uma lei de equilíbrio. Se tivermos muito controle das alterações, criaremos problemas. Se tivermos pouco controle, criaremos outros problemas.

Em um grande projeto de software, alterações não controladas levam rapidamente ao caos. Para projetos assim, o controle de alterações combina procedimentos humanos e ferramentas automatizadas, proporcionando um mecanismo para o controle de alterações. O processo de controle de alterações está ilustrado esquematicamente na Figura 22.5. Uma *solicitação de alteração* é apresentada e avaliada para determinar o mérito técnico, efeitos colaterais potenciais, impacto global sobre outros objetos de configuração e funções do sistema e o custo projetado da alteração. Os resultados da avaliação são apresentados como um *relatório de alterações*, usado por uma *autoridade de controle de alterações* (*change control authority* — CCA) — uma pessoa ou grupo de pessoas que toma a decisão final sobre o status e prioridade da alteração. Uma *ordem de alteração de engenharia* (*engineering change order* - ECO) é gerada para cada alteração aprovada. A ECO descreve a alteração a ser feita, as restrições que devem ser respeitadas e o critério para revisar e auditar.

Os objetos a ser alterados podem ser colocados em um diretório que é controlado apenas pelo engenheiro de software que está fazendo a alteração. Um sistema de controle de versão (ver o quadro *Ferramentas de software* sobre CVS) atualiza o arquivo original logo que a alteração foi feita. Como alternativa, os objetos a ser alterados podem ser “retirados” do banco de dados do projeto (repositório), as alterações ser feitas e aplicadas às atividades SQA apropriadas. Os objetos são então “colocados” no banco de dados e usados mecanismos de controle de versão apropriados (Seção 22.3.2) para criar a próxima versão do software.

Esses mecanismos de controle de versão, integrados ao processo de controle de alterações, implementam dois elementos importantes da gestão de alterações — controle de acesso e controle de sincronização. O *controle de acesso* controla quais os engenheiros de software têm autoridade para acessar e modificar um objeto de configuração particular. O *controle de sincronização* ajuda a assegurar que alterações paralelas, executadas por duas pessoas diferentes, não sobrescrevam uma à outra.

Você pode se sentir incomodado pelo nível de burocracia gerado pela descrição do processo de controle de alteração mostrado na Figura 22.5. Essa sensação não é incomum. Sem as

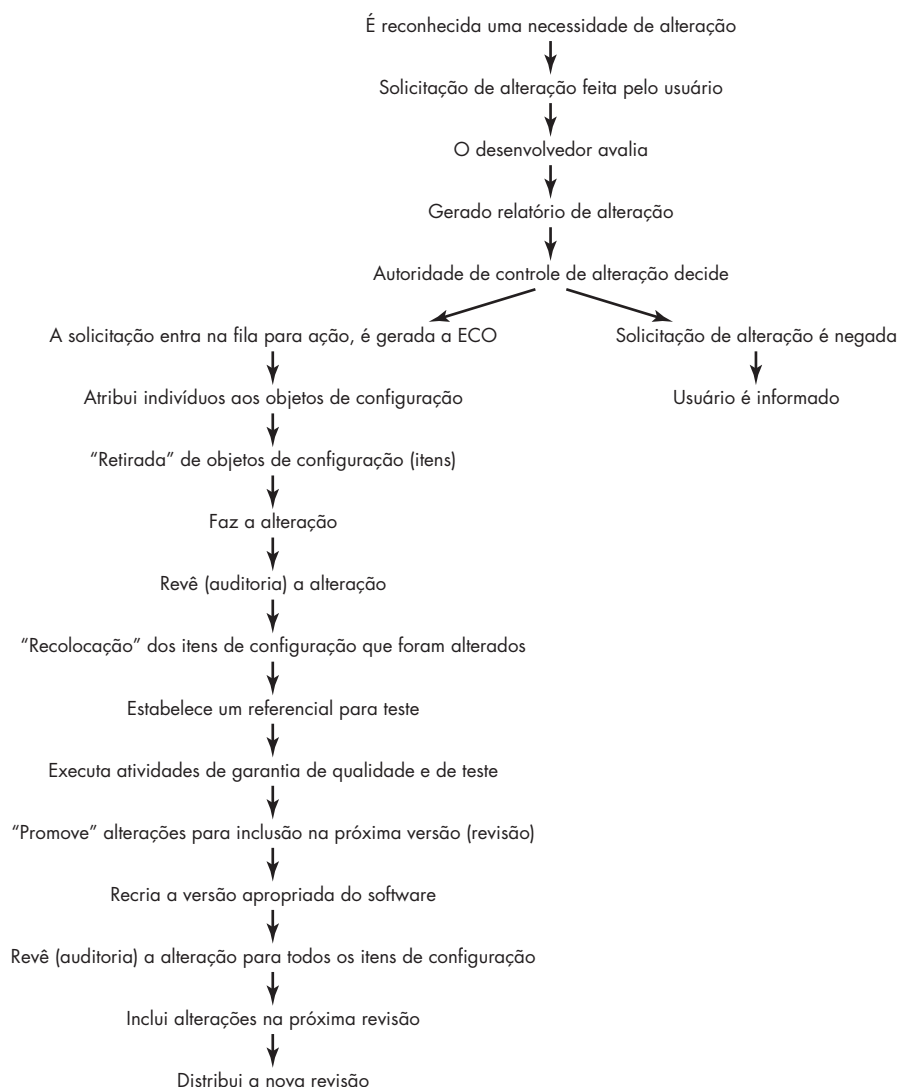
“A arte do progresso é preservar a ordem nas alterações e preservar as alterações na ordem.”

Alfred North Whitehead

PONTO-CHAVE

Deve-se observar que muitas solicitações de alterações podem ser combinadas para resultar em uma única ECO e que ECOs resultam tipicamente em alterações a múltiplos objetos de configuração.

FIGURA 22.5

O processo de controle de alterações

condições de segurança apropriadas, o controle de alterações pode retardar o progresso e criar barreiras desnecessárias. Grande parte dos desenvolvedores de software que usam mecanismos de controle de alterações (infelizmente, muitos não usam nenhum) já criou uma série de camadas de controle para ajudar a evitar os problemas mencionados aqui.



AVISO
 Opte por um pouco mais de controle de alterações do que você acha que precisará. É provável que a dose certa seja bem maior.

Antes de uma SCI se tornar um referencial, só é necessário usar o *controle informal de alteração*. O desenvolvedor do objeto de configuração (SCI) em questão pode fazer todas as alterações justificáveis pelo projeto e pelos requisitos técnicos (desde que as alterações não afetem requisitos mais amplos do sistema que estejam fora do escopo de trabalho do desenvolvedor). Uma vez que o objeto tenha passado pela revisão técnica e tenha sido aprovado, pode ser criado um referencial.⁵ Uma vez que uma SCI se torna um referencial, é implementado o *controle de alterações em nível de projeto*. Agora, para fazer uma alteração, o desenvolvedor precisa ter novamente a aprovação do gerente de projeto (se a alteração for “local”) ou da CCA se a alteração afetar outras SCIs. Em alguns casos, são necessárias solicitações formais de geração de altera-

⁵ Um referencial pode ser criado por outras razões também. Por exemplo, quando são criadas “construções diárias”, todos os componentes verificados por determinado tempo se tornam o referencial para o trabalho do dia seguinte.

ções, relatórios de alterações e ECOs. No entanto, é feita a avaliação de cada alteração e todas as alterações são acompanhadas e revisadas.

Quando o artefato de software é liberado para os clientes, institui-se o *controle formal de alterações*. O procedimento formal de controle de alterações foi resumido na Figura 22.5.

A autoridade de controle de alterações desempenha um papel ativo no segundo e terceiro níveis de controle. Dependendo do tamanho e do tipo de projeto de software, a CCA pode ser composta por uma pessoa — o gerente de projeto — ou um grupo de pessoas (por exemplo, representantes do software, hardware, engenharia do banco de dados, suporte, marketing). O papel da CCA é assumir uma visão global, isto é, avaliar o impacto das alterações além da SCI em questão. Como a alteração afetará o hardware? Como a alteração afetará o desempenho? Como a alteração modificará a percepção do cliente em relação ao produto? Como a alteração afetará a qualidade e a confiabilidade do produto? Essas e muitas outras questões são resolvidas pela CCA.

“O troco é inevitável, exceto para as máquinas automáticas de refrigerantes.”

Adesivo em para-choque

22.3.4 Auditoria de configuração

Identificação, controle de versão e controle de alterações ajudam a manter a ordem naquilo que de outra forma seria uma situação caótica. No entanto, mesmo os melhores mecanismos de controle rastreiam uma alteração somente até que seja gerada uma ECO. Como a equipe de software pode assegurar que a alteração foi implementada corretamente? A resposta é dupla: (1) revisões técnicas e (2) a auditoria de configuração de software.

A revisão técnica (Capítulo 15) focaliza a exatidão técnica do objeto de configuração modificado. Os revisores avaliam a SCI para determinar a consistência com outras SCIs, omissões ou efeitos colaterais potenciais. Deverá ser feita uma revisão técnica para todas as alterações, exceto as mais triviais.

Uma *auditoria de configuração de software* complementa a revisão técnica avaliando o objeto de configuração quanto a características que em geral não são consideradas durante a revisão. A auditoria propõe e responde as seguintes questões:

CASA SEGURA



Problemas SCM

Cena: Escritório de Doug Miller no início do projeto de software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe de engenharia de artefato de software.

Conversa:

Doug: Eu sei que ainda é cedo para isso, mas precisamos falar sobre gestão de alterações.

Vinod (rindo): Dificilmente. O pessoal de marketing ligou esta manhã e eles tinham algumas “segundas intenções”. Nada importante, mas é só o começo.

Jamie: Nós fomos muito informais a respeito de gestão de alterações em projetos anteriores.

Doug: Eu sei, mas este é maior e mais visível, e pelo que me lembro...

Vinod (balançando a cabeça): Nos matamos por alterações descontroladas no projeto de controle de luz ambiental... Lembro-me dos atrasos...

Doug (franzindo a testa): Um pesadelo que eu prefiro não lembrar.

Jamie: Então o que fazemos?

Doug: Penso que devemos fazer três coisas. Primeiro temos que desenvolver — ou tomar emprestado — um processo de controle de alterações.

Jamie: Você quer dizer, o modo como as pessoas solicitam alterações?

Vinod: Sim, mas também como avaliamos a alteração, como decidimos quem faz (se é que nós decidimos sobre isso) e como mantemos os registros do que é afetado pela alteração.

Doug: Em segundo lugar, precisamos realmente arranjar uma boa ferramenta de SCM para controle de versão e alteração.

Jamie: Podemos criar um banco de dados para todos os nossos artefatos.

Vinod: Nesse contexto, elas são chamadas de SCIs, e há muitas boas ferramentas que proporcionam suporte para isso.

Doug: É um bom começo, agora temos que...

Jamie: Ei, Doug, você disse que eram três coisas...

Doug (sorrindo): Terceiro — todos nós temos que seguir os processos de gestão de alterações e usar as ferramentas — não importa o quê, OK?

? Quais as perguntas primárias que fazemos durante uma auditoria de configuração?

1. Foi feita a alteração especificada na ECO? Alguma modificação adicional foi incorporada?
2. Foi feita uma revisão técnica para avaliar a exatidão técnica?
3. Seguiu-se o processo do software e os padrões de engenharia de software foram aplicados adequadamente?
4. A alteração foi “destacada” no SCI? A data e o autor da alteração foram especificados? Os atributos do objeto de configuração refletem a alteração?
5. Seguiram-se os procedimentos da SCM para anotar a alteração, registrá-la e relatá-la?
6. Todos os SCIs relacionados foram adequadamente atualizados?

Em alguns casos, as perguntas de auditoria são formuladas como parte da revisão técnica. No entanto, quando a SCM é uma atividade formal, a auditoria de configuração é conduzida separadamente pelo grupo de garantia de qualidade. Essas auditorias de configuração formais também asseguram que os SCIs corretos (por versão) tenham sido incorporados em uma construção específica e que toda a documentação esteja atualizada e consistente com a versão construída.

22.3.5 Relatório de status

O relatório de status de configuração (às vezes chamado de contabilidade de status) é uma tarefa da SCM que responde às seguintes questões: (1) O que aconteceu? (2) Quem fez? (3) Quando aconteceu? (4) O que mais será afetado?

O fluxo de informações para o relatório de status de configuração (CSR) está ilustrado na Figura 22.5. A cada vez que a um SCI é atribuída uma identificação nova ou atualizada, faz-se uma entrada no CSR. Cada vez que uma alteração é aprovada pela CCA (isto é, é gerada uma ECO), é feita uma entrada no CSR. Cada vez que se executa uma auditoria de configuração, os resultados são relatados como parte da tarefa do CSR. A saída do CSR pode ser colocada em um banco de dados on-line ou em um site, de forma que os desenvolvedores de software ou pessoal de suporte possam acessar as informações de alterações por categoria de palavra-chave. Além disso, é gerado um relatório do CSR regularmente; ele se destina a manter a gerência e os profissionais informados sobre alterações importantes.



Desenvolva uma lista do tipo “precisa saber” para todo objeto de configuração e mantenha-a atualizada. Quando é feita uma alteração, certifique-se de que todos os que estão na lista sejam notificados.

FERRAMENTAS DO SOFTWARE



Suporte de SCM

Objetivo: as ferramentas de SCM proporcionam suporte para uma ou mais das atividades de processo discutidas na Seção 22.3.

Mecânica: muitas ferramentas de SCM modernas funcionam em conjunto com um repositório (um sistema de banco de dados) e proporcionam mecanismos para identificação, versão e controle de alterações, auditoria e relatórios.

Ferramentas representativas:⁶

CCC/Harvest, distribuída pela Computer Associates (www.cai.com), é um sistema de SCM multiplataforma.

ClearCase, desenvolvida pela Rational, proporciona uma família de funções de SCM (www-306.ibm.com/software/awdtools/clearcase/index.html).

Serena ChangeMan ZMF, distribuída pela Serena (www.serena.com/US/products/zmf/index.aspx),

contém um conjunto completo de ferramentas de SCM aplicáveis tanto ao software convencional quanto às WebApps.

SourceForge, distribuída pela VA Software (sourceforge.net), contém gerenciamento de versão, capacidades de construção, rastreamento de problema/bug e muitas outras características de gestão.

SurroundSCM, desenvolvida pela Seapine Software, proporciona recursos completos de gestão de alterações (www.seapine.com).

Vesta, distribuída pela Compac, é um sistema de SCM de domínio público que pode suportar projetos pequenos (<10 KLOC) e grandes (10.000 KLOC) (www.vestasys.org).

Uma lista bem organizada de ferramentas e ambientes de SCM comerciais pode ser encontrada em www.cmtoday.com/yp/commercial.html.

⁶ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

22.4 GESTÃO DE CONFIGURAÇÃO PARA WEBAPPS

Neste livro, é discutida a natureza especial das aplicações para Web e de métodos especializados (chamados de *métodos de engenharia Web*⁷) necessários para criá-las. Dentre as muitas características que diferenciam as WebApps do software tradicional está a natureza onipresente da alteração.

 **Que impacto uma alteração não controlada tem sobre uma WebApp?**

Os desenvolvedores para WebApp muitas vezes usam um modelo de processo iterativo, incremental, que aplica muitos princípios derivados do desenvolvimento ágil de software (Capítulo 3). Por meio dessa abordagem, uma equipe de engenharia muitas vezes desenvolve um incremento para WebApp em um período de tempo muito curto usando uma abordagem focada no cliente. Incrementos subsequentes adicionam conteúdo e funcionalidade, e cada um deles tende a implementar alterações que levam a um conteúdo aperfeiçoado, melhor utilização, melhor estética, melhor navegação, melhor desempenho e maior segurança. Portanto, no mundo ágil das WebApps, a alteração é vista de forma um tanto diferente.

Se você é um membro de uma equipe para WebApp, tem de adotar as alterações. E ainda mais, uma equipe ágil típica evita todas as coisas que parecem ser intensivas que tornam processo pesado burocrático e formal. A gestão de configuração de software é vista com frequência (embora incorretamente) como detentora dessas características. Essa contradição é remediada, não pela rejeição dos princípios, práticas e ferramentas de SCM, mas sim moldando-as para satisfazerem às necessidades especiais dos projetos de WebApp.

22.4.1 Problemas dominantes

Na medida em que as WebApps se tornam cada vez mais importantes para a sobrevivência e crescimento dos negócios, crescem as necessidades da gestão de configuração. Por quê? Porque sem controles eficazes, alterações impróprias a uma WebApp (lembre-se de que o imediatismo e a evolução contínua são os atributos dominantes de muitas WebApps) podem levar a: uma colocação não autorizada de informações sobre novos produtos, funcionalidade errônea ou mal testada que causa frustração nos visitantes de um site, brechas na segurança que põem em risco os sistemas internos da empresa e outras consequências economicamente desagradáveis ou até mesmo desastrosas.

As estratégias gerais para gestão de configuração de software (SCM) descritas neste capítulo são aplicáveis, mas as táticas e as ferramentas devem ser adaptadas para se conformarem com a natureza especial das WebApps. Quatro aspectos [Dar99] deverão ser considerados ao desenvolvermos táticas para gestão de configuração de WebApp.

Conteúdo. Uma WebApp típica contém um vasto conjunto de conteúdo — texto, gráficos, applets, scripts, arquivos de áudio/vídeo, formulários, elementos de página ativos, tabelas, dados encadeados e muitos outros. O desafio é organizar esse mar de conteúdo em um conjunto racional de objetos de configuração (Seção 22.1.4) e, então, estabelecer mecanismos de controle de configuração apropriados para esses objetos. Uma abordagem é modelar o conteúdo da WebApp usando técnicas convencionais de modelagem de dados (Capítulo 6), anexando um conjunto de propriedades especializadas a cada objeto. A natureza estática/dinâmica de cada objeto e sua longevidade projetada (por exemplo, objeto temporário, de existência fixa ou permanente) são exemplos de propriedades necessárias para estabelecer uma abordagem de SCM eficaz. Por exemplo, se um item de conteúdo é alterado a cada hora, ele tem longevidade temporária. Os mecanismos de controle para esse item seriam diferentes (menos formais) daqueles aplicados a um componente de formulários que é um objeto permanente.

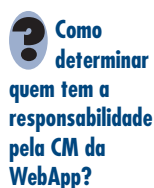
Pessoas. Devido ao fato de que uma porcentagem significativa do desenvolvimento de WebApp continua a ser executada de maneira *ad hoc*, qualquer pessoa envolvida na WebApp pode criar conteúdo (e frequentemente o faz). Muitos criadores de conteúdo não possuem conhecimentos

7 Ver em [Pre08] uma discussão simples dos métodos de engenharia para Web.

em engenharia de software e ignoram completamente a necessidade de gestão de configuração. Consequentemente, a aplicação cresce e é alterada de maneira não controlada.

Escalabilidade. As técnicas e controles aplicados a uma pequena WebApp não são bem escaláveis. Não é raro uma simples WebApp crescer significativamente enquanto são implementadas interconexões com sistemas de informação existentes, bancos de dados, armazém de dados e gateways de portais. À medida que cresce o tamanho e a complexidade, pequenas mudanças podem ter efeitos amplos e inesperados que podem se tornar problemáticos. Portanto, o rigor dos mecanismos de configuração deverá ser diretamente proporcional à escala de aplicação.

Políticas. Quem é o “dono” de uma WebApp? Essa é uma pergunta feita em empresas grandes e pequenas, e sua resposta tem um impacto significativo sobre as atividades de gerenciamento e controle. Em alguns casos os desenvolvedores para Web estão instalados fora da organização de TI, criando dificuldades potenciais de comunicação. Dart [Dar99] sugere as seguintes perguntas para ajudar a entender as políticas associadas com engenharia para Web:



- Quem assume a responsabilidade pela exatidão das informações no site?
- Quem garante que os processos de controle de qualidade foram obedecidos antes que as informações fossem publicadas no site?
- Quem é responsável por fazer alterações?
- Quem assume o custo da alteração?

As respostas a essas perguntas ajudam a determinar as pessoas na organização que devem adotar um processo de gestão de configuração para WebApps.

A gestão de configuração para WebApps continua a evoluir (por exemplo, [Ngu06]). Um processo de SCM convencional pode ser muito desajeitado, mas uma nova geração de *ferramentas de gestão de conteúdo* especificamente projetadas para a engenharia surgiu nos últimos anos. Essas ferramentas estabelecem um processo que adquire as informações existentes (de uma ampla variedade de objetos WebApp), gerencia as alterações nos objetos, estrutura essas alterações para que possam ser apresentadas a um usuário final e as apresenta ao ambiente do lado do cliente para ser exibidas.

22.4.2 Objetos de configuração de WebApp

As WebApps abrangem grande variedade de objetos de configuração — objetos de conteúdo (por exemplo, texto, gráficos, imagens, vídeo e áudio), componentes funcionais (por exemplo, scripts, applets) e objetos de interface (por exemplo, COM ou CORBA). Os objetos da WebApp podem ser identificados (podem ser atribuídos nomes de arquivo para eles) de qualquer forma que seja apropriada para a organização. No entanto, recomendam-se as seguintes convenções para assegurar que seja mantida a compatibilidade entre plataformas: nomes de arquivos deverão ser limitados a 32 caracteres, deverão ser evitados nomes de arquivos com misturas de caracteres maiúsculos e minúsculos ou nomes com todas as letras em maiúscula e deverá ser evitado também o uso de underlines em nomes de arquivos. Além disso, referências a URL (links) dentro de um objeto de configuração devem sempre usar caminhos relativos (por exemplo, ../products/alarmsensors.html).

Todo o conteúdo da WebApp tem formato e estrutura. Os formatos de arquivos internos são ditados pelo ambiente de computação no qual o conteúdo está armazenado. No entanto, o *formato de renderização* (muitas vezes chamado de *formato de exibição*) é definido pelo estilo estético e regras de design estabelecidas para WebApp. A *estrutura de conteúdo* define uma arquitetura de conteúdo; ela define a maneira pela qual são montados os objetos de conteúdo para apresentar informações claras ao usuário final. Boiko [Boi04] define estrutura como “mapas que você coloca sobre um conjunto de conteúdo [objetos] para organizá-los e torná-los acessíveis às pessoas que precisam deles”.

22.4.3 Gestão de conteúdo

“Gestão de conteúdo é um antídoto para o emaranhado de informações de hoje.”

Bob Boiko

A *gestão de conteúdo* está relacionada com a gestão de configuração no sentido de que um sistema de gestão de conteúdo (CMS) estabelece um processo (suportado por ferramentas apropriadas) que adquire o conteúdo existente (de uma ampla variedade de objetos de configuração de WebApp), estrutura esse conteúdo de maneira que ele possa ser apresentado a um usuário final e, então, fornece-o ao ambiente no lado do cliente para ser exibido.

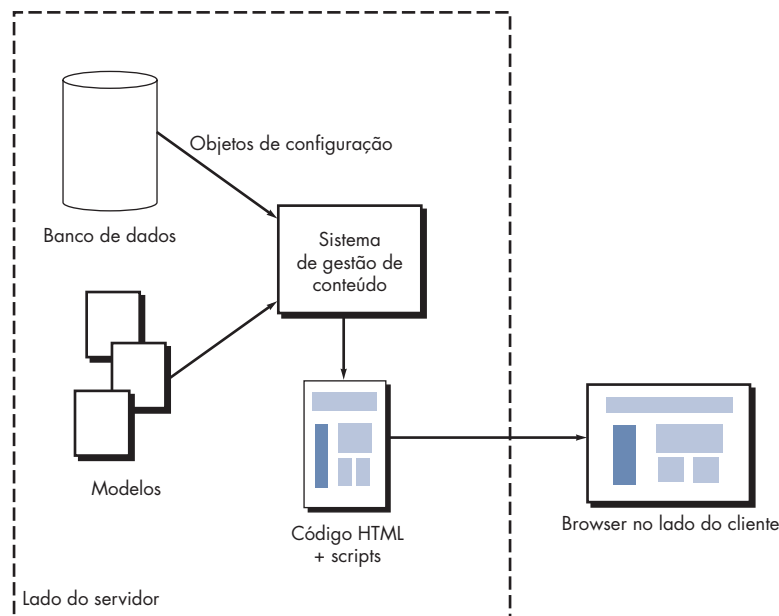
O uso mais comum de um sistema de gestão de conteúdo ocorre quando é criada uma WebApp dinâmica. As WebApps dinâmicas criam páginas Web “dinamicamente”. Isto é, o usuário tipicamente consulta a WebApp solicitando informações específicas. A WebApp consulta um banco de dados, formata as informações corretamente e as apresenta ao usuário. Por exemplo, uma empresa de música fornece uma biblioteca de CDs para venda. Quando o usuário solicita um CD ou seu arquivo de música equivalente, é consultado um banco de dados, e uma variedade de informações sobre o artista, o CD (por exemplo, sua imagem ou elementos gráficos), o conteúdo musical e uma amostra de áudio, tudo isso é baixado (download) e configurado em um modelo de conteúdo padrão. A página Web resultante é criada no lado do servidor e passada para o lado do browser cliente para ser examinada pelo usuário final. Uma representação genérica disso está na Figura 22.6.

No sentido mais geral, um CMS “configura” conteúdo para o usuário final por meio da invocação de três subsistemas integrados: um subsistema de coleção, um subsistema de gestão e um subsistema de publicação [Boi04].

O subsistema de coleção. O conteúdo é extraído dos dados e de informações que devem ser criados ou adquiridos por um gerador de conteúdo. O *subsistema de coleção* abrange todas as ações necessárias para criar e/ou adquirir conteúdo e as funções técnicas que são necessárias para (1) converter conteúdo de maneira que possa ser representado por uma linguagem de marcação (por exemplo, HTML, XML), e (2) organiza o conteúdo em pacotes que podem ser efetivamente mostrados no lado do cliente.

FIGURA 22.6

Sistema de gestão de conteúdo



PONTO-CHAVE

O subsistema de coleção abrange todas as ações necessárias para adquirir, e/ou converter conteúdo de maneira que possa ser apresentado no lado do cliente.

PONTO-CHAVE

O subsistema de gestão implementa um repositório para todo o conteúdo. A gestão de configuração é executada nesse subsistema.

Criação e aquisição de conteúdo (também chamada de *autoria*) ocorre muitas vezes em paralelo com outras atividades de desenvolvimento de WebApp e em geral é conduzido por criadores de conteúdo não técnicos. Essa atividade combina elementos de criatividade e pesquisa e é suportada por ferramentas que permitem ao autor do conteúdo caracterizá-lo de modo que possa ser padronizado para uso dentro da WebApp.

Uma vez existindo o conteúdo, ele deve ser convertido para se adaptar aos requisitos de um CMS. Isso implica eliminar o conteúdo bruto de quaisquer informações desnecessárias (por exemplo, representações gráficas redundantes), formatação do conteúdo para se adaptar aos requisitos do CMS e mapear os resultados em uma estrutura de informações que permita que seja gerenciado e publicado.

O subsistema de gestão. Uma vez existindo o conteúdo, ele deve ser armazenado em um repositório, catalogado para aquisição e uso subsequente, e rotulado para definir (1) status atual (por exemplo, o objeto de conteúdo está completo ou em desenvolvimento?), (2) a versão apropriada do objeto de conteúdo, e (3) objetos de conteúdo relacionados. Portanto, o *subsistema de gestão* implementa um repositório que abrange os seguintes elementos:

- *Banco de dados de conteúdo* — a estrutura de informações estabelecida para armazenar todos os objetos de conteúdo.
- *Recursos de banco de dados* — funções que permitem ao CMS pesquisar objetos de conteúdo específicos (ou categorias de objetos), armazenar e recuperar objetos e gerenciar a estrutura de arquivo estabelecida para o conteúdo.
- *Funções de gestão de configuração* — os elementos funcionais e o workflow associado que suporta identificação do objeto de conteúdo, controle de versão, gestão de alterações, gestão de auditoria e relatos.

Além desses elementos, o subsistema de gestão implementa uma função de administração que abrange os metadados e regras que controlam a estrutura global do conteúdo e a maneira pela qual ele é suportado.

PONTO-CHAVE

O subsistema de publicação extrai o conteúdo do repositório e o fornece aos navegadores do lado do cliente.

O subsistema de publicação. O conteúdo deve ser extraído de um repositório, convertido para uma forma que seja conveniente para a publicação e formatado de maneira que possa ser transmitido aos navegadores do lado do cliente. O subsistema de publicação executa essas tarefas usando uma série de modelos (templates). Cada *modelo* é uma função que cria uma publicação por meio de um dentre três componentes diferentes [Boi04]:

- *Elementos estáticos* — texto, gráficos, mídia e scripts que não requerem outros processamentos são transmitidos diretamente para o lado do cliente.
- *Serviços de publicação* — chamadas de função para serviços específicos de acesso e formatação que personalizam o conteúdo (usando regras predefinidas), executam a conversão dos dados e criam links de navegação apropriados.
- *Serviços externos* — fornecem acesso à infraestrutura de informação corporativa externa como, por exemplo, aplicações de dados ou de retaguarda da empresa.

Um subsistema de gestão de conteúdo que abrange cada um desses subsistemas é aplicável à maior parte dos projetos para WebApp. No entanto, a filosofia e funcionalidade básicas associadas com um CMS são aplicáveis a todas as WebApps dinâmicas.

22.4.4 Gestão de alterações

O fluxo de trabalho associado ao controle de alterações para software convencional (Seção 22.3.3) em geral é muito ponderado para desenvolvimento de WebApp. É pouco provável que a solicitação de alteração, relato da alteração e sequência de ordem de mudança de engenharia possam ser conseguidos de forma ágil que seja aceitável para a maioria dos projetos de desenvolvimento de WebApp. Como podemos então controlar uma corrente contínua de alterações solicitadas para o conteúdo e funcionalidade da WebApp?

FERRAMENTAS DO SOFTWARE

**Gestão de conteúdo**

Objetivo: ajudar os engenheiros de software e criadores de conteúdo na gestão de conteúdo que é incorporado nas WebApps.

Mecânica: ferramentas nesta categoria permitem aos engenheiros da Web e criadores de conteúdo atualizar o conteúdo da WebApp de forma controlada. Muitos estabelecem um simples sistema de gestão de arquivo que atribui permissões de atualizações página por página para vários tipos de conteúdo da WebApp. Outros mantêm um sistema de controle de versões para que uma versão anterior de um conteúdo possa ser arquivada para fins históricos.

Ferramentas representativas:⁸

Vignette Content Management, desenvolvida pela Vignette (www.vignette.com/us/Products), é um conjunto de ferramentas de gestão de conteúdo corporativo.

ektron-CMS300, desenvolvida pela ektron (www.ektron.com), é um conjunto de ferramentas que proporciona recursos de gestão de conteúdo, como também ferramentas de desenvolvimento para Web.

OmniUpdate, desenvolvida pela WebsiteASP, Inc. (www.omniupdate.com), é uma ferramenta que permite que os provedores de conteúdo autorizado desenvolvam atualizações controladas para conteúdo específico da WebApp.

Informações adicionais sobre SCM e ferramentas de gestão de conteúdo para engenharia para Web podem ser encontradas em um ou mais dos seguintes sites: *Web Developer's Virtual Encyclopedia* (www.wdlv.com), *WebDeveloper* (www.webdeveloper.com), *Developer Shed* (www.devshed.com), *webknowhow.net* (www.webknowhow.net) ou *WebReference* (www.webreference.com).

Para implementar um gerenciamento efetivo de alterações segundo a filosofia de “codifique e vá em frente” que continua a dominar o desenvolvimento de WebApp, o processo convencional de controle de alterações deve ser modificado. Cada alteração deverá ser classificada em uma dentre quatro classes:

Classe 1 — alteração de conteúdo ou função que corrige um erro ou melhora o conteúdo ou a funcionalidade local.

Classe 2 — alteração de conteúdo ou função que tenha impacto sobre outros objetos de conteúdo ou sobre os componentes funcionais.

Classe 3 — alteração de conteúdo ou função que tenha um amplo impacto através de uma WebApp (por exemplo, extensão ou funcionalidade principais, melhora significativa ou redução em conteúdo, alterações importantes necessárias na navegação).

Classe 4 — alteração importante de projeto (por exemplo, alteração na abordagem do projeto da interface ou na abordagem de navegação) que será notada imediatamente por uma ou mais categorias de usuário.

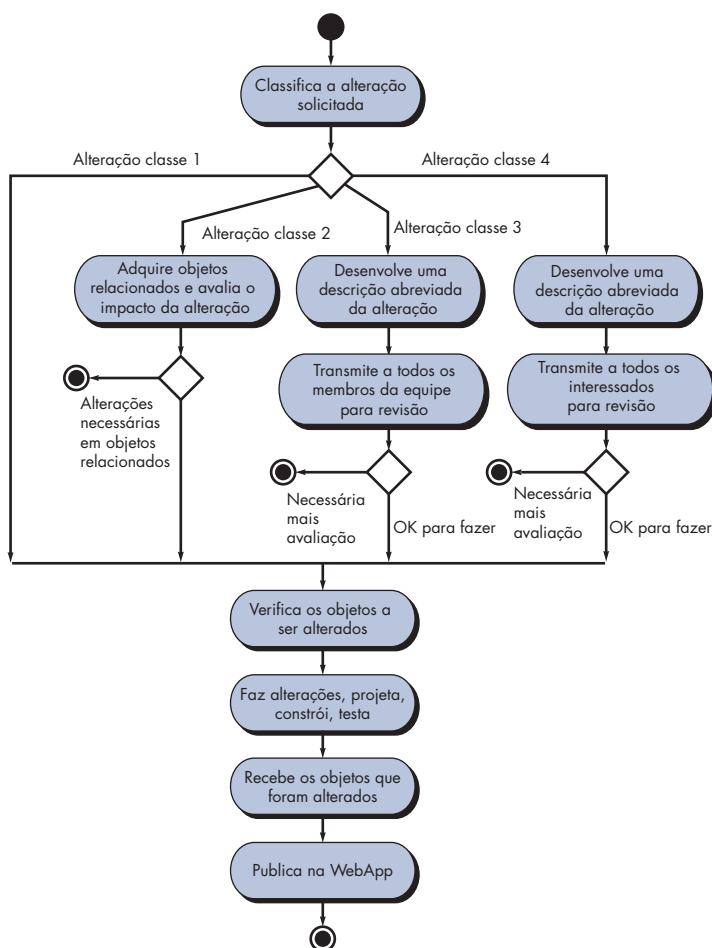
Uma vez classificada a solicitação de alteração, ela pode ser processada de acordo com o algoritmo mostrado na Figura 22.7.

De acordo com a figura, as alterações classe 1 e classe 2 são tratadas informalmente e são manipuladas de modo ágil. Para uma alteração classe 1, você avaliaria o impacto da mudança, mas não é necessária nenhuma revisão externa ou documentação. À medida que a alteração é feita, os procedimentos-padrão de entrada (check-in) e saída (check-out) são apoiados por ferramentas de repositório de configuração. Para alterações classe 2, você deveria revisar o impacto da alteração sobre objetos relacionados (ou pedir a outros desenvolvedores responsáveis por aqueles objetos que o façam). Se a alteração pode ser feita sem necessidade de alterações significativas em outros objetos, a modificação ocorre sem revisão ou documentação adicional. Se forem necessárias alterações substanciais, mais avaliação e planejamento serão exigidos.

Alterações classe 3 e 4 também são tratadas de uma forma ágil, mas é necessária alguma documentação descritiva e procedimentos de revisão mais formais. Para as alterações classe 3 é desenvolvida uma *descrição de alteração* — descrevendo a alteração e fornecendo uma breve avaliação do impacto da alteração. A descrição é distribuída a todos os membros da equipe que a

⁸ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

FIGURA 22.7

Gerenciando alterações para WebApps**FERRAMENTAS DO SOFTWARE****Gestão de alterações**

Objetivo: ajudar os projetistas da Web e criadores de conteúdo na gestão de alterações à medida que são feitas nos objetos de configuração para WebApp.

Mecânica: as ferramentas desta categoria foram desenvolvidas originalmente para software convencional, mas podem ser adaptadas para ser usadas pelos engenheiros da Web e criadores de conteúdo para fazer alterações controladas nas WebApps. Elas suportam entrada e saída automáticas, controle de versão e desfazer alterações (rollback), relatos e outras funções da SCM.

Ferramentas representativas:⁹

ChangeMan WCM, desenvolvida pela Serena (www.serena.com), é um conjunto de ferramentas de gestão de alteração que proporcionam recursos de SCM completos.

ClearCase, desenvolvida pela Rational (www-306.ibm.com/software/rational/sw-atoz/indexC.html), é um conjunto de ferramentas que proporcionam recursos completos de gestão de configuração para WebApps.

Source Integrity, desenvolvida pela mks (www.mks.com), é uma ferramenta de SCM que pode ser integrada a ambientes de desenvolvimento selecionados.

examinam para melhor avaliar seu impacto. É desenvolvida também uma descrição de alteração para as alterações classe 4, mas nesse caso a revisão é conduzida por todos os interessados.

⁹ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

22.4.5 Controle de versão

À medida que uma WebApp evolui por meio de uma série de incrementos, podem existir várias versões diferentes ao mesmo tempo. Uma versão (a WebApp operacional corrente) está disponível na Internet para os usuários finais; outra versão (o próximo incremento da WebApp) pode estar nos estágios finais de teste antes da distribuição/instalação (*deployment*); uma terceira versão está em desenvolvimento e representa uma grande atualização em conteúdo, estética de interface e funcionalidade. Objetos de configuração podem estar claramente definidos para que cada um possa ser associado à versão apropriada. Além disso, devem ser estabelecidos mecanismos de controle. Dreilinger [Dre99] discute a importância do controle de versão (e alteração) quando escreve:

Em um site *não controlado* no qual múltiplos autores têm acesso para editar e contribuir, surge um potencial para conflitos e problemas — mais ainda quando esses autores trabalham em locais diferentes em horários diferentes do dia e da noite. Você pode passar o dia melhorando o arquivo *index.html* para um cliente. Depois que você fez as suas alterações, outro desenvolvedor que trabalha em casa após o horário comercial ou em outro escritório, pode gastar a noite fazendo o uploading de sua própria nova versão revisada do arquivo *index.html*, sobrescrevendo completamente o seu trabalho sem maneira de recuperá-lo!

É provável que você já tenha passado por uma situação assim. Para tanto, é necessário um processo de controle de versão.

1. *Deverá ser estabelecido um repositório central para o projeto de WebApp.* O repositório terá as versões correntes de todos os objetos de configuração de WebApp (conteúdo, componentes funcionais e outros).
2. *Cada projetista da Web cria sua própria pasta de trabalho.* A pasta contém aqueles objetos que estão sendo criados ou alterados em determinado instante.
3. *Os relógios das estações de trabalho de todos os desenvolvedores deverão estar sincronizados.* Isso é feito para evitar conflitos de sobrescrita quando dois desenvolvedores fazem alterações com horários muito próximos.
4. *Na medida em que novos objetos de configuração são desenvolvidos ou objetos existentes são alterados, eles são importados para o repositório central.* A ferramenta de controle de versão (veja discussão sobre CVS na barra lateral) irá gerenciar todas as funções de check-in (entrada) e check-out (saída) das pastas de trabalho de cada desenvolvedor da WebApp. A ferramenta também fornecerá atualizações automáticas de e-mail a todas as partes interessadas quando forem feitas alterações no repositório.
5. *À medida que objetos são importados ou exportados do repositório, é gerada uma mensagem automática com data e hora.* Isso proporciona informações úteis para auditoria e pode se tornar parte de um esquema eficaz de relatórios.

A ferramenta de controle de versão mantém diferentes versões da WebApp e pode reverter para uma versão mais antiga se necessário.

22.4.6 Auditoria e relatório

Para melhorar a agilidade, as funções de auditoria e relatório não são enfatizadas no trabalho de engenharia Web¹⁰. No entanto, elas não são todas eliminadas. Todos os objetos que entram (check-in) ou saem (check-out) do repositório são registrados em um log (registro) que pode ser revisto quando se desejar. Pode ser criado um relatório completo de forma que todos os membros da equipe da WebApp tenham uma cronologia das alterações durante um período definido. Além disso, uma notificação automática por e-mail (endereçada a todos os desenvolvedores e interessados que tenham interesse) pode ser enviada todas as vezes que um objeto entra ou sai do repositório.

¹⁰ Isso está começando a mudar. Há uma ênfase cada vez maior no SCM como um elemento da segurança WebApp [Sar06]. Proporcionando um mecanismo para rastrear e relatar todas as alterações feitas em um objeto WebApp, uma ferramenta de gestão de alterações pode proporcionar uma valiosa proteção contra alterações mal-intencionadas.

INFORMAÇÕES

**Normas de SCM**

A seguir é apresentada uma lista de normas de SCM (extraída em parte do site www.12207.com) razoavelmente abrangente:

IEEE Standards standards.ieee.org/catalog/olis/

IEEE 828 Software Configuration Management Plans

IEEE 1042 Software Configuration Management

ISO Standards www.iso.ch/iso/en/ISOOnline.frontpage

ISO 10007:1995 Quality Management, Guidance for CM

ISO/IEC 12207 Information Technology-Software Life Cycle Processes

ISO/IEC TR 15271 Guide for ISO/IEC 12207

ISO/IEC TR 15846 Software Engineering-Software Life Cycle Process-Configuration Management for Software Order

EIA Standards www.eia.org/

EIA 649 National Consensus Standard for Configuration Management

EIA CMB4-1A Configuration Management Definitions for Digital Computer Programs

EIA CMB4-2 Configuration Identification for Digital Computer Programs

EIA CMB4-3 Computer Software Libraries

EIA CMB4-4 Configuration Change Control for Digital Computer Programs

EIA CMB6-1C Configuration and Data Management References Order

EIA CMB6-3 Configuration Identification

EIA CMB6-4 Configuration Control

EIA CMB6-5

Textbook for Configuration Status Accounting

EIA CMB7-1

Electronic Interchange of Configuration Management Data

U.S. Military Standards

Information of MIL standards: www-library.itsi.disa.mil

DoD MIL STD-973

Configuration Management

MIL-HDBK-61

Configuration Management Guidance

Outras normas

DO-178B

Guidelines for the Development of Aviation Software

ESA PSS-05-09

Guide to Software Configuration Management

AECL CE-1001-STD

rev.1 Standard for Software Engineering of Safety Critical Software

DOE SCM checklist:

<http://cio.doe.gov/ITReform/sqse/download/cmcklst.doc>

BS-6488

British Std., Configuration Management of Computer-Based Systems

Best Practice — UK

Office of Government Commerce: www.ogc.gov.uk

CMII

Institute of CM Best Practices: www.icmhq.com

Um guia de recursos de gestão de configuração (*Configuration Management Resource Guide*) proporciona informações complementares para aqueles interessados nos processos e prática de gestão de alterações (CM). O guia está disponível no site www.quality.org/config/cm-guide.html.

22.5 RESUMO

A gestão de configuração de software (SCM) é uma atividade abrangente aplicada em todo o processo de software. A SCM identifica, controla, faz auditoria e relata modificações que invariavelmente ocorrem enquanto o software está sendo desenvolvido e depois que foi entregue ao cliente. Todos os produtos criados como parte da engenharia de software tornam-se parte de uma configuração de software. A configuração é organizada de maneira que permite controle ordenado das alterações.

A configuração de software é composta por uma série de objetos inter-relacionados, também chamados de itens de configuração de software (SCIs), que são produzidos como resultado de alguma atividade de engenharia de software. Além dos documentos, programas e dados, o ambiente de desenvolvimento usado para criar software também pode ser colocado sob o controle de configuração. Todas os SCIs são armazenados em um repositório que implementa uma série de mecanismos e estruturas de dados para assegurar a integridade dos dados, proporcionar suporte de integração para outras ferramentas de software, suportar compartilhamento de informações entre todos os membros da equipe de software e implementar funções no suporte do controle de versão e alteração.

Uma vez desenvolvido e revisado um objeto de configuração, ele se torna uma referência. Alterações em um objeto referencial resultam na criação de uma nova versão daquele objeto.

A evolução de um programa pode ser acompanhada examinando-se o histórico de revisão de todos os objetos de configuração. O controle de versão é uma série de procedimentos e ferramentas para gerenciar o uso desses objetos.

O controle de alteração é uma atividade procedimental que assegura qualidade e consistência quando são feitas alterações em um objeto de configuração. O processo de controle de alterações começa com uma solicitação de alteração, leva a uma decisão sobre fazer ou rejeitar a solicitação de alteração e culmina com uma atualização controlada do SCI que deve ser alteada.

A auditoria de configuração é uma atividade de SQA que ajuda a assegurar que a qualidade seja mantida quando feitas alterações. Os relatórios de status fornecem informações sobre cada alteração para aqueles que precisam ter conhecimento do assunto.

A gestão de configuração para WebApps é similar em muitos aspectos à de SCM para software convencional. No entanto, cada uma das tarefas centrais de SCM deverá ser agilizada para torná-la o mais leve possível e devem ser implementadas provisões especiais para gestão de conteúdo.

PROBLEMAS E PONTOS A PONDERAR

22.1. Por que a Primeira Lei da Engenharia de Software é verdadeira? Forneça exemplos específicos para cada uma das quatro razões fundamentais para alterações.

22.2. Quais são os quatro elementos que existem quando é implementado um sistema de SCM eficaz? Discuta rapidamente cada um.

22.3. Discuta as razões para referenciais (baselines) com suas próprias palavras.

22.4. Suponha que você seja o gerente de um pequeno projeto. Que referenciais definiria para o projeto e como os controlaria?

22.5. Desenvolva um sistema de banco de dados de projeto (repositório) que permitiria a um engenheiro de software armazenar, estabelecer referências cruzadas, acompanhar, atualizar e alterar todos os itens importantes da configuração de software. Como o banco de dados trataria com diferentes versões do mesmo programa? O código-fonte seria tratado de forma diferente da documentação? Como dois desenvolvedores seriam impedidos de fazer alterações diferentes no mesmo SCI ao mesmo tempo?

22.6. Pesquise uma ferramenta de SCM existente e descreva como ela implementa o controle para versões, variantes e objetos de configuração em geral.

22.7. As relações <parte-de> e <inter-relacionado> representam relações simples entre objetos de configuração. Descreva cinco relações adicionais que podem ser úteis no contexto de um repositório de SCM.

22.8. Pesquise uma ferramenta de SCM existente e descreva como ela implementa o mecanismo do controle de versão. Como alternativa, leia duas ou três publicações sobre a SCM e descreva as diferentes estruturas de dados e mecanismos de referência usados para o controle de versão.

22.9. Desenvolva uma lista de checagem (*checklist*) para usar durante as auditorias de configuração.

22.10. Qual é a diferença entre uma auditoria de SCM e uma revisão técnica? Podem suas funções serem incluídas em uma revisão? Quais são os prós e os contras?

22.11. Descreva rapidamente as diferenças entre a SCM para software convencional e a SCM para WebApps.

22.12. O que é gestão de conteúdo? Use a Web para pesquisar as características de uma ferramenta de gestão de conteúdo e forneça um breve resumo.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Entre as ofertas mais recentes sobre SCM estão Leon (*Software Configuration Management Handbook*, 2d ed., Artech House Publishers, 2005), Maraia (*The Build Master: Microsoft's Software Configuration Management Best Practices*, Addison-Wesley, 2005), Keyes (*Software Configuration Management*, Auerbach, 2004) e Hass (*Configuration Management Principles and Practice*, Addison-Wesley, 2002). Cada um desses livros apresenta todo o processo de SCM com detalhes substanciais. Maraia (*Software Configuration Management Implementation Roadmap*, Wiley, 2004) fornece um guia do tipo “como fazer para...” para aqueles que devem implementar a SCM em uma organização. Lyon (*Practical CM*, Raven Publishing, 2003, disponível em www.configuration.org) descreveu um guia simples para o profissional de CM incluindo diretrizes pragmáticas para implementar todos os aspectos de um sistema de gestão de configuração (atualizado anualmente). White e Clemm (*Software Configuration Management Strategies and Rational ClearCase*, Addison-Wesley, 2000) apresentam a SCM dentro do contexto de uma ou mais ferramentas populares de SCM.

Berczuk e Appleton (*Software Configuration Management Patterns*, Addison-Wesley, 2002) mostram uma variedade de padrões úteis que ajudam a entender a SCM e a implementar sistemas de SCM eficazes. Brown et al. (*Anti-Patterns and Patterns in Software Configuration Management*, Wiley, 1999) discutem o que não se deve fazer (antipadrões) ao implementar um processo de SCM e, em seguida, trata dos remédios. Bays (*Software Release Methodology*, Prentice-Hall, 1999) focaliza o mecanismo de “versão bem-sucedida de produtos”, um complemento importante para uma SCM eficaz.

Conforme as WebApps se tornam mais dinâmicas, a gestão de conteúdo tem se tornado um tópico essencial para engenheiros da Web. Livros de White (*The Content Management Handbook*, Curtin University Books, 2005), Jenkins e seus colegas (*Enterprise Content Management Methods*, Open Text Corporation, 2005), Boiko [Boi04], Mauthe e Thomas (*Professional Content Management Systems*, Wiley, 2004), Addey e seus colegas (*Content Management Systems*, Glasshaus, 2003), Rockley (*Managing Enterprise Content*, New Riders Press, 2002), Hackos (*Content Management for Dynamic Web Delivery*, Wiley, 2002) e Nakano (*Web Content Management*, Addison-Wesley, 2001) apresentam bons tratamentos do assunto.

Além das discussões do tópico, Lim e seus colegas (*Enhancing Microsoft Content Management Server with ASP.NET 2.0*, Packt Publishing, 2006), Ferguson (*Creating Content Management Systems in Java*, Charles River Media, 2006), IBM Redbooks (*IBM Workplace Web Content Management for Portal 5.1 and IBM Workplace Web Content Management 2.5*, Vivante, 2006), Fritz e seus colegas (*Typo3: Enterprise Content Management*, Packt Publishing, 2005) e Forta (*Reality ColdFusion: Intranets and Content Management*, Pearson Education, 2002) abordam gestão de conteúdo no contexto de ferramentas e linguagens específicas.

Uma ampla variedade de fontes de informação sobre gestão de configuração de software e gestão de conteúdo está disponível na Internet. Uma lista atualizada das referências na Web, relevantes à gestão de configuração de software, pode ser encontrada no site www.mhhe.com/engcs/compSci/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

indicador539

Meta/Questão/Métrica
(Goal/Question/Metric -
GQM).....541

medida538

medição539

métricas

atributos das ...542

código-fonte ...559

modelo

de requisitos....543

orientado a classe .551

projeto da
arquitetura....547projeto da interface
de usuário557projeto orientado
a objeto549projeto para
WebApp557

Um elemento-chave de qualquer processo de engenharia é a medição. Você pode usar medidas para melhorar o entendimento dos atributos dos modelos criados e para avaliar a qualidade dos produtos ou sistemas construídos. Mas diferentemente de outras disciplinas de engenharia, a engenharia de software não é fundamentada nas leis quantitativas da física. Medidas diretas, como tensão (voltagem), massa, velocidade ou temperatura, são incomuns no mundo do software. Devido às medidas e métricas de software serem muitas vezes indiretas, elas estão abertas ao debate. Fenton [Fen91] trata desse assunto quando afirma:

Medição é o processo pelo qual números ou símbolos são anexados aos atributos de entidades no mundo real para defini-los de acordo com regras claramente estabelecidas... Nas ciências físicas, medicina, economia e mais recentemente nas ciências sociais, podemos medir os atributos antes considerados incomensuráveis... É claro que essas medidas não são tão refinadas como muitas feitas nas ciências físicas, mas existem [e são tomadas decisões importantes baseadas nelas]. Percebemos que a obrigação de tentar “medir o incomensurável” para melhorar nossa compreensão de entidades particulares é tão poderosa na engenharia de software quanto em qualquer outra disciplina.

Mas alguns membros da comunidade de software continuam a argumentar que o software é “incomensurável” ou que tentativas de medições deverão ser adiadas até entendermos melhor o software e os atributos que deverão ser usados para descrevê-lo. Isso é um erro.

PANORAMA

O que é? Por sua natureza, a engenharia é uma disciplina quantitativa. A métrica de produto ajuda os engenheiros de software a visualizar o projeto e a construção do software, focalizando atributos específicos e mensuráveis dos artefatos da engenharia de software.

Quem realiza? Os engenheiros de software usam métricas de produto para ajudá-los a criar software de mais alta qualidade.

Por que é importante? Haverá sempre um elemento qualitativo na criação de software. O problema é que a avaliação qualitativa pode não ser suficiente. É preciso critérios objetivos para ajudar a direcionar o projeto de dados, arquitetura, interfaces e componentes. Ao testar, necessitamos de orientação quantitativa que nos auxiliará na seleção de casos de teste e seus objetivos. A métrica de produto proporciona uma base por meio da qual a análise, projeto, codificação e teste podem ser conduzidos mais objetivamente e avaliados de maneira mais quantitativa.

Quais são as etapas envolvidas? O primeiro passo no processo de medição é derivar as

medições de software e as métricas apropriadas para a representação do software. Em seguida, coletam-se os dados necessários para derivar as métricas formuladas. Uma vez computadas, as métricas apropriadas são analisadas com base em diretrizes preestabelecidas e dados do passado. Os resultados das análises são interpretados para obter informações sobre a qualidade do software e os dados da interpretação levam à modificação dos requisitos e modelos de projeto, código-fonte ou casos de teste. Em algumas instâncias, pode também levar à modificação do próprio processo de software.

Qual é o artefato? As métricas de produto computadas por meio de dados coletados dos requisitos e modelos de projeto, código-fonte e casos de teste.

Como garantir que o trabalho foi realizado corretamente? Devemos estabelecer os objetivos da medição antes de iniciarmos a coleta de dados, definindo cada métrica de produto de maneira não ambígua. Defina apenas algumas métricas e então as use para obter informações sobre a qualidade de um artefato de software.

teste	560
ponto de função (FP)	543
princípios de medição	540

Embora as métricas de produto para programas de computadores sejam imperfeitas, podem proporcionar uma maneira sistemática de avaliar a qualidade com base em um conjunto de regras claramente definidas. Elas também proporcionam uma visão objetiva, que “vai direto ao ponto” e não “após o fato”. Isso permite descobrir e corrigir problemas potenciais antes que se tornem defeitos catastróficos.

Neste capítulo, apresentam-se medidas que podem ser usadas para avaliar a qualidade do produto enquanto está sendo projetado. Essas medidas de atributos internos do produto fornecem uma indicação em tempo real da eficácia dos modelos de requisitos, projeto e código; a eficácia dos casos de teste; e a qualidade geral do software que será criado.

23.1 ESTRUTURA PARA MÉTRICAS DE PRODUTO

“Uma ciência é tão desenvolvida quanto suas ferramentas de medição.”

Louis Pasteur

Conforme mencionado na introdução, a medição atribui números ou símbolos a atributos de entidades no mundo real. Para tanto, é necessário um modelo de medição abrangendo um conjunto consistente de regras. Embora a teoria da medição (por exemplo, [Kyb84]) e sua aplicação a programas de computadores (por exemplo, [Zus97]) sejam tópicos que estão além do escopo deste livro, vale estabelecer uma estrutura fundamental e um conjunto de princípios básicos que orientem a definição de métricas de produto para software.

23.1.1 Medidas, métricas e indicadores

? Qual a diferença entre uma medida e uma métrica?

Embora os termos *medida*, *medição* e *métricas* sejam usados com frequência de forma intercambiável, é importante notar as diferenças sutis entre eles. Como *medida* pode ser usada como substantivo ou como verbo, as definições do termo podem se tornar confusas. Sob o contexto de engenharia de software, *medida* proporciona uma indicação quantitativa da extensão, quantidade, capacidade ou tamanho de algum atributo de um produto ou processo. *Medição* é o ato de determinar uma medida. O *IEEE Standard Glossary of Software Engineering Terminology* [IEE93b] define *métrica* como “uma medida quantitativa do grau com o qual um sistema, componente ou processo possui determinado atributo.”

Quando é coletado um único ponto de dado (por exemplo, o número de erros descobertos em um componente de software), foi estabelecida uma medida. A medição ocorre como resultado da coleção de um ou mais pontos de dados (por exemplo, um conjunto de revisões de componente e testes de unidade são investigados para coletar medidas do número de erros para cada um). Uma métrica de software relaciona as medidas individuais de alguma maneira (por exemplo, o número médio de erros encontrados por revisão ou o número médio de erros encontrados por teste de unidade).

Um engenheiro de software coleta medidas e desenvolve métricas para obter indicadores. Um *indicador* é uma métrica ou combinação de métricas que proporcionam informações sobre o processo de software, em um projeto de software ou no próprio produto. Um indicador proporciona informações que permitem ao gerente de projeto ou aos engenheiros de software ajustar o processo, o projeto ou o produto para incluir melhorias.

23.1.2 O desafio das métricas de produto

Durante as últimas quatro décadas, muitos pesquisadores tentaram desenvolver uma única métrica que forneça uma medida fácil de entender a complexidade do software. Fenton [Fen94] caracteriza essa pesquisa como uma busca do “impossível santo graal”. Embora tenham sido propostas dezenas de medidas de complexidade [Zus90], cada uma delas tem uma visão diferente do que é a complexidade e quais os atributos de um sistema que levam à complexidade. Por analogia, considere uma métrica para avaliar um carro de luxo. Alguns observadores podem enfatizar o projeto da carroceria; outros considerar as características mecânicas; outros ainda podem considerar o custo ou desempenho ou o uso de combustíveis alternativos ou a facilidade

PONTO-CHAVE

Um indicador é uma métrica ou métricas que proporcionam visão do processo, produto ou do projeto.

“Assim como a medição da temperatura começou com um dedo indicador... E chegou até as escalas, instrumentos e técnicas sofisticadas, a medição de software também está evoluindo.”

Shari Pfleeger

de reciclagem quando o carro já estiver inutilizável. Como cada uma dessas características pode ser estranha em relação às outras, é difícil derivar um valor único para “atraente”. O mesmo problema ocorre com o software.

No entanto, há uma necessidade de medir e controlar a complexidade do software. E, se é difícil obter um valor único dessa métrica de qualidade, deverá ser possível desenvolver medidas de diferentes atributos internos de programa (por exemplo, modularidade efetiva, independência funcional e outros discutidos no Capítulo 8). Essas medidas e métricas derivadas dos atributos podem ser usadas como indicadores independentemente da qualidade dos modelos de requisitos e projeto. Mas aqui também temos problemas. Fenton [Fen94] observa isso quando diz: “O perigo de tentar encontrar medidas que caracterizem tantos atributos diferentes é que inevitavelmente as medidas têm de satisfazer interesses em conflito. Isso é contrário à teoria representacional da medição”. Embora a afirmação de Fenton esteja correta, muitos argumentam que a medição de produto executada durante os primeiros estágios do processo de software fornece aos engenheiros um mecanismo consistente e objetivo para avaliar a qualidade.

É justo questionar, no entanto, quão válidas são as métricas de produto. Quanto as métricas de produto estão alinhadas à confiabilidade e qualidade de longo prazo para um sistema de computador? Fenton [Fen91] trata a questão da seguinte maneira:

Apesar das conexões intuitivas entre a estrutura interna dos artefatos [métricas de produto] e seus atributos externos de produto e processo, tem havido na realidade poucas tentativas científicas de estabelecer relações específicas. Há várias razões para isso; a mais citada é a impraticabilidade de executar experimentos relevantes.

Cada um dos “desafios” mencionados aqui é motivo para cuidado, mas não há por que dispensar as métricas de produto.¹ A medição é essencial se o que se deseja é qualidade.

23.1.3 Princípios de medição

Antes de introduzirmos uma série de métricas de produto que (1) ajuda na avaliação dos modelos de análise e projeto, (2) proporciona uma indicação da complexidade dos projetos procedimentais e código-fonte e (3) facilita o projeto de testes mais eficazes, é importante entendermos os princípios básicos de medições. Roche [Roc94] sugere um processo de medição que pode ser caracterizado por cinco atividades:

- *Formulação.* A criação de medidas e métricas de software apropriadas para a representação do software considerado.
- *Coleção.* O mecanismo usado para acumular dados necessários para criar as métricas formuladas.
- *Análise.* A computação das métricas e a aplicação de ferramentas matemáticas.
- *Interpretação.* A avaliação de métricas que resultam em informações sobre a qualidade da representação.
- *Feedback.* Recomendações derivadas da interpretação de métricas de produto transmitidas para a equipe de software.

As métricas de software serão úteis somente se forem efetivamente caracterizadas e validadas de forma que demonstrem valer a pena. Os princípios a seguir [Let03b] são representativos de muitos que podem ser propostos para caracterização métrica e validação:

- *Uma métrica deve ter as propriedades matemáticas desejadas.* O valor da métrica deverá estar em um intervalo significativo (por exemplo, 0 a 1, em que 0 significa ausência, 1 indica

WebRef

Um grande volume de informações sobre métricas de produto foi compilado por Horst Zuse em irb.cs.tu-berlin.de/~zuse/.

? Quais os passos para um processo eficaz de medição?

¹ Embora seja comum na literatura a crítica a métricas específicas, muitas críticas focalizam aspectos secretos e perdem o objetivo primário das métricas no mundo real: ajudar o engenheiro de software a estabelecer uma maneira sistemática e objetiva de visualizar seu trabalho e melhorar a qualidade do produto como resultado.



Na realidade, muitas métricas de produto em uso atualmente não se enquadram nesses princípios tão bem quanto deveriam. Mas isso não significa que elas não têm valor — apenas seja cuidadoso ao usá-las, entendendo que se destinam a proporcionar informações, não uma rígida verificação científica.

o valor máximo e 0,5 representa o “ponto médio”). Além disso, uma métrica que deve estar em uma escala racional não deve ser composta de componentes medidos apenas em escala ordinal.

- Quando uma métrica representa uma característica do software que aumenta na ocorrência de peculiaridades positivas ou diminui na ocorrência de peculiaridades indesejadas, o valor da métrica deverá aumentar ou diminuir da mesma maneira.
- Cada métrica deverá ser validada empiricamente em uma ampla variedade de contextos antes de ser publicada ou usada para tomada de decisões. Uma métrica deverá medir o fator de interesse, independentemente de outros fatores. Ela deve ser “ampliada” para sistemas grandes e funcionar em uma variedade de linguagens de programação e domínios de sistemas.

Embora a formulação, caracterização e validação sejam de extrema importância, a coleta e a análise são as atividades que regem o processo de medição. Roche [Roc94] sugere os seguintes princípios para essas atividades: (1) sempre que possível, a coleta e análise de dados deverá ser automática; (2) deverão ser aplicadas técnicas estatísticas válidas para estabelecer relações entre atributos internos de produto e características externas de qualidade (por exemplo, se o nível de complexidade da arquitetura está correlacionado ao número de defeitos relatados em uso); e (3) diretrizes e recomendações interpretativas deverão ser estabelecidas para cada métrica.

23.1.4 Medição de software orientada a objetivo

O paradigma *Meta/Questão/Métrica* (*Goal/Question/Metric* – GQM) foi desenvolvido por Basili e Weiss [Bas84] como uma técnica para identificar métricas significativas para qualquer parte do processo de software. O GQM enfatiza a necessidade de (1) estabelecer um *objetivo* de medição explícita que é específico para a atividade do processo ou característica de produto que deve ser avaliada, (2) definir um conjunto de *questões* que devem ser respondidas para atingir o objetivo e (3) identificar *métricas* bem formuladas que ajudam a responder a essas questões.

Pode ser usado um *modelo de definição de objetivo* (*goal definition template*) [Bas94] para definir o objetivo de cada medição. O modelo toma a seguinte forma:

Analisar {o nome da atividade ou atributo a ser medido} **com a finalidade de** {o objetivo geral da análise²} **com relação a** {o aspecto da atividade ou atributo considerado} **do ponto de vista de** {a pessoa que tem o interesse na medição} **no contexto de** {o ambiente no qual a medição ocorre}.

Como exemplo, considere um modelo de definição de objetivo para o *CasaSegura*:

Analisar a arquitetura de software do *Casa Segura* **com a finalidade de** avaliar componentes da arquitetura **com relação à** habilidade em tornar o *CasaSegura* mais ampliável **do ponto de vista dos** engenheiros de software que estão executando o trabalho **no contexto de** aperfeiçoamento de produto durante os próximos três anos.

Com o objetivo de medição explicitamente definido, desenvolve-se uma série de questões. As respostas a essas questões ajudam a equipe de software (ou outros interessados) a determinar se o objetivo de medição foi atingido. Entre as questões que podem ser formuladas estão:

- Q₁: Os componentes de arquitetura são caracterizados para distinguir função de dados relacionados?
- Q₂: A complexidade de cada componente está dentro dos limites que facilitarão a modificação e extensão?

WebRef

Uma discussão útil sobre GQM pode ser encontrada em www.thedacs.com/GoldPractices/practices/gqma.html.

² van Solingen e Berghout [Sol99] sugerem que o objetivo é quase sempre “entender, controlar ou improvisar” a atividade de processo ou atributo de produto.

Cada uma dessas questões deverá ser respondida quantitativamente, usando uma ou mais medidas e métricas. Por exemplo, uma métrica que proporciona uma indicação da coesão (Capítulo 8) de um componente de arquitetura pode ser útil na resposta de Q_1 . Métricas discutidas mais adiante neste capítulo podem proporcionar uma visão para Q_2 . Em todos os casos, as métricas escolhidas (ou derivadas) deverão estar de acordo com os princípios de medição discutidos na Seção 23.1.3 e os atributos de medição discutidos na Seção 23.1.5.

23.1.5 Atributos de métricas eficazes de software

Centenas de métricas já foram propostas para programas de computadores, mas nem todas proporcionam suporte prático ao engenheiro de software. Algumas demandam medições muito complexas, outras são tão esotéricas que poucos profissionais do mundo real têm qualquer esperança de entendê-las, e outras ainda violam as noções intuitivas básicas do que é realmente um software de alta qualidade.

Ejiogu [Eji91] define um conjunto de atributos que deverão ser abrangidos por métricas, de software efetivas. A métrica derivada e as medições que levam até ela deverão ser:

 **Como devemos avaliar a qualidade de uma métrica de software proposta?**

- *Simples e computáveis.* Deverá ser relativamente fácil aprender a derivar a métrica, e sua computação não deve demandar esforço ou tempo fora do normal.
- *Empiricamente e intuitivamente persuasiva.* A métrica deverá satisfazer as ideias do engenheiro sobre o atributo do produto considerado (por exemplo, uma métrica que mede coesão de módulo deverá crescer em valor na medida em que aumenta o nível de coesão).
- *Consistente e objetiva.* A métrica deverá sempre produzir resultados que não sejam ambíguos. Um terceiro independente deverá ser capaz de derivar o mesmo valor da métrica usando as mesmas informações sobre o software.
- *Consistente no seu uso das unidades e dimensões.* A computação matemática da métrica deverá usar medidas que não resultem em combinações bizarras de unidades. Por exemplo, multiplicar número de pessoas nas equipes de projeto pelas variáveis da linguagem de programação no programa resulta em uma mistura duvidosa de unidades que não é claramente convincente.
- *Independente da linguagem de programação.* As métricas deverão ser baseadas no modelo de requisitos, modelo de projeto ou na própria estrutura do programa. Elas não deverão ser dependentes dos caprichos da sintaxe ou semânticas das linguagens de programação.
- *Um mecanismo efetivo para feedback de alta qualidade.* A métrica deverá fornecer informações que podem levar a um produto final de melhor qualidade.



A experiência indica que a métrica de um produto será usada somente se ela for clara e fácil de computar. Se forem necessárias dezenas de “contagens” e computações complexas, é pouco provável que seja amplamente adotada.

Embora muitas métricas de software satisfaçam a esses atributos, algumas usadas comumente podem não satisfazer a um ou outro desses atributos. Um exemplo é o ponto de função (*function point* – FP) (discutida na Seção 23.2.1) – uma medida da “funcionalidade” fornecida pelo software. Alguém pode argumentar³ que o atributo consistente e objetivo falha porque um terceiro, independente, pode não ser capaz de derivar o mesmo valor ponto de função que um colega usando as mesmas informações sobre o software. Deveremos então rejeitar a medida FP? A resposta é “claro que não!” A métrica FP proporciona informações úteis e valor distinto, mesmo que não satisfaça a um atributo perfeitamente.

3 Pode-se apresentar um contra-argumento igualmente forte. Essa é a natureza da métrica de software.

CASA SEGURA



Debatendo métricas de produto

Cena: Sala do Vinod.

Atores: Vinod, Jamie e Ed — membros da equipe de engenharia de software do CasaSegura dando segmento ao trabalho de projeto em nível de componente e projeto de casos de teste.

Conversa:

Vinod: Doug [Doug Miller, gerente de engenharia de software] me disse que todos nós devemos usar métricas de produto, mas ele foi um pouco vago. Ele falou também que não vai “forçar a barra”... Usar ou não, depende de nós.

Jamie: Ótimo, porque não tenho como arranjar tempo para começar a medir coisas. Estamos lutando para manter o cronograma.

Ed: Concordo com o Jamie. Somos contra, aqui... Não temos tempo.

Vinod: Sim, eu sei, mas provavelmente deve haver algum mérito em usar tais métricas.

Jamie: Não estou discutindo isso, Vinod, é a questão do tempo... E nenhum de nós tem qualquer disponibilidade.

Vinod: Mas se essas medições ajudarem a poupar o seu tempo?

Ed: Errado, demandam horas livres e como disse o Jamie...

Vinod: Não, espere... E se isso nos ajudar a poupar tempo?

Jamie: Como?

Vinod: Retrabalho... É isso. Se uma medição que fizermos nos ajudar a evitar um problema maior ou mesmo moderado, e se isso impedir o retrabalho em uma parte do sistema, nos poupará tempo. Não?

Ed: É possível, suponho, mas você pode garantir que alguma métrica de produto nos ajudará a encontrar um problema?

Vinod: Você pode garantir que não?

Jamie: Então o que você propõe?

Vinod: Acho que poderíamos selecionar algumas métricas de projeto, provavelmente orientadas a classe, e usá-las como parte de nosso processo de revisão para qualquer componente que desenvolvermos.

Ed: Não estou familiarizado com métricas orientadas a classe.

Vinod: Vou verificar tudo isso e fazer uma recomendação... Ok com vocês pessoal?

[Ed e Jamie balançam a cabeça sem muito entusiasmo.]

23.2 MÉTRICAS PARA O MODELO DE REQUISITOS

O trabalho técnico na engenharia de software começa com a criação do modelo de requisitos. É nesse estágio que os requisitos são formulados e é estabelecida uma base para o projeto. Portanto, métricas de produto que proporcionem informações sobre a qualidade do modelo de análise são desejáveis.

Embora relativamente poucas métricas de análise e especificação tenham aparecido na literatura, é possível adaptar as usadas frequentemente para estimativa de projeto e aplicá-las nesse contexto. Essas métricas examinam o modelo de requisitos com a intenção de prever o “tamanho” do sistema resultante. O tamanho é às vezes (mas nem sempre) um indicador da complexidade do projeto e quase sempre é um indicador do trabalho cada vez maior de codificação, integração e teste.

23.2.1 Métricas baseadas em função

A *métrica ponto de função* pode ser usada efetivamente como um meio para medir a funcionalidade fornecida por um sistema.⁴ Por meio de dados históricos, a métrica FP pode ser empregada para (1) estimar o custo ou trabalho necessário para projetar, codificar e testar o software; (2) prever o número de erros que serão encontrados durante o teste; e (3) prever o número de componentes e/ou o número de linhas projetadas de código-fonte no sistema implementado.

Pontos de função são derivados por meio de uma relação empírica baseada em medidas calculáveis (diretas) do domínio de informações do software e avaliações qualitativas da complexidade do software. Valores do domínio de informações são definidos da seguinte maneira:⁵

WebRef

Muitas informações úteis sobre ponto de função podem ser obtidas em www.ifpug.org e www.functionpoints.com.

⁴ Centenas de livros, dissertações e artigos já foram escritos sobre métricas FP. Uma boa bibliografia pode ser encontrada em [IFP05].

⁵ Atualmente, a definição dos valores do domínio de informações e a maneira pela qual elas são contadas é algo um pouco mais complexo. O leitor interessado deve consultar [IFP01] para mais detalhes.

Número de entradas externas (*number of external inputs – EEs*). Cada *entrada externa* é originada de um usuário ou transmitida de outra aplicação e fornece dados distintos orientados a aplicação ou informações de controle. Entradas são muitas vezes usadas para atualizar *arquivos lógicos internos* (*internal logical files – ILFs*). As entradas devem ser diferenciadas das consultas, que são contadas separadamente.

Número de saídas externas (*number of external outputs – EOs*). Cada *saída externa* é formada por dados derivados da aplicação e fornece informações para o usuário. Nesse contexto, as saídas externas se referem a relatórios, telas, mensagens de erro etc. Itens individuais de dados em um relatório não são contados separadamente.

Número de consultas externas (*number of external inquiries – EQs*). Uma *consulta externa* é definida como uma entrada on-line que resulta na geração de alguma resposta imediata do software na forma de uma saída on-line (muitas vezes obtida de um ILF).

Número de arquivos lógicos internos (*number of internal logical files – ILFs*). Cada *arquivo lógico interno* é um agrupamento lógico de dados que reside dentro das fronteiras do aplicativo e é mantido através de entradas externas.

Número de arquivos de interface externos (*number of external interface files – EIFs*). Cada *arquivo de interface externo* é um agrupamento lógico de dados que reside fora da aplicação, mas fornece informações que podem ser usadas pela aplicação.

Uma vez coletados esses dados, é completada a tabela da Figura 23.1 e associado um valor de complexidade com cada contagem. Organizações que usam métodos ponto de função desenvolvem critérios para determinar se determinada entrada é simples, média ou complexa. No entanto, a determinação da complexidade é de certo modo subjetivo.

Para calcular pontos de função (FP), usa-se a seguinte relação:

$$FP = \text{contagem total} \times [0,65 + 0,01 \times \sum (F_i)] \quad (23.1)$$

em que a contagem total é a soma de todas as entradas FP obtidas da Figura 23.1.

Os F_i ($i = 1$ a 14) são *fatores de ajuste de valor* (*value adjustment factors – VAF*) baseados em respostas às questões a seguir [Lon02]:

1. O sistema requer salvamento (*backup*) e recuperação confiável (*recovery*)?
2. São necessárias comunicações de dados especializadas para transferir informações para a aplicação ou da aplicação?
3. Há funções de processamento distribuído?

PONTO-CHAVE

Os fatores de ajuste de valor são usados para fornecer uma indicação da complexidade do problema.

FIGURA 23.1

Computando pontos de função

Valor do Domínio de Informação	Contagem		Fator de peso			
			Simple	Médio	Complexo	
Entradas externas (EIs)		×	3	4	6	=
Saídas Externas (EOs)		×	4	5	7	=
Consultas Externas (EQs)		×	3	4	6	=
Arquivos Lógicos Internos (ILFs)		×	7	10	15	=
Arquivos de Interface Externos (EIFs)		×	5	7	10	=
Contagem total						→

4. O desempenho é crítico?
5. O sistema rodará em um ambiente operacional existente e intensamente utilizado?
6. O sistema requer entrada de dados on-line?
7. A entrada on-line de dados requer que a transação de entrada seja composta em múltiplas telas ou operações?
8. Os ILFs são atualizados on-line?
9. As entradas, saídas, arquivos ou consultas são complexas?
10. O processamento interno é complexo?
11. O código é projetado para ser reutilizável?
12. A conversão e instalação estão incluídas no projeto?
13. O sistema é projetado para múltiplas instalações em diferentes organizações?
14. A aplicação é projetada para facilitar a troca e o uso pelo usuário?

WebRef

Há uma calculadora on-line para FP em irb.cs.unimagdeburg.de/sw-eng/us/java/fp/.

Cada uma dessas perguntas é respondida por meio de uma escala que varia de 0 (não importante ou não aplicável) a 5 (absolutamente essencial). Os valores das constantes na Equação (23.1) e os fatores de peso aplicados aos valores do domínio de informações são determinados empiricamente.

Para ilustrarmos o uso da métrica FP nesse contexto, consideramos a representação de um modelo simples de análise, apresentado na Figura 23.2. Na figura, está representado um diagrama de fluxo de dados (Capítulo 7) para uma função do software *CasaSegura*.

A função gerencia a interação do usuário, aceitando uma senha para ativar ou desativar o sistema e possibilita consultas sobre o estado das zonas de segurança e vários sensores de segurança. A função mostra uma série de mensagens imediatas e envia sinais de controle apropriados para os vários componentes do sistema de segurança.

O diagrama de fluxo de dados é avaliado para determinar um conjunto-chave de medidas de domínio de informação necessárias para a computação da métrica ponto de função. Três entradas externas — **senha**, **botão de emergência** e **ativar/desativar** — são exibidas na figura com duas consultas externas — **consulta de zona** e **consulta de sensor**. É mostrado um ILF (**arquivo de configuração de sistema**). São apresentadas também duas saídas externas (**mensagens** e **estado do sensor**) e quatro EIFs (**sensor de teste**, **configuração de zona**, **ativar/desativar** e **alerta de alarme**). Esses dados, com sua complexidade apropriada, estão na Figura 23.3.

O total da contagem mostrado na Figura 23.3 deve ser ajustado usando a Equação (23.1). Para as finalidades desse exemplo, supomos que $\sum (F_i)$ é 46 (um produto moderadamente complexo). Portanto,

$$FP = 50 \times [0,65 \times (0,01 \times 46)] = 56$$

FIGURA 23.2

Um modelo de fluxo de dados para o software *CasaSegura*

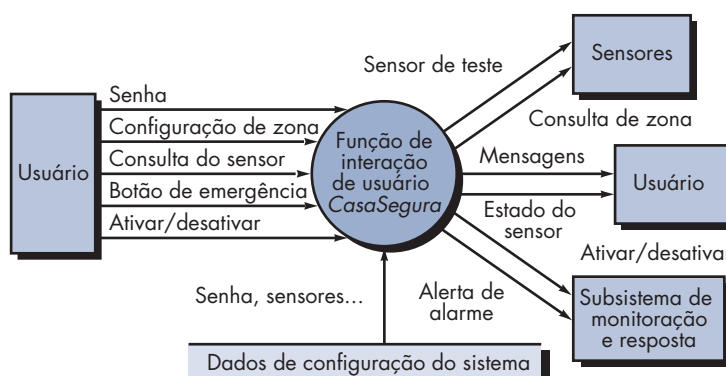


FIGURA 23.3

Computando pontos de função

Valor do Domínio de Informação	Contagem		Fator de peso			
			Simples	Médio	Complexo	
Entradas externas (EIs)	3	×	3	4	6	= 9
Saídas Externas (EOs)	2	×	4	5	7	= 8
Consultas Externas (EQs)	2	×	3	4	6	= 6
Arquivos Lógicos Internos (ILFs)	1	×	7	10	15	= 7
Arquivos de Interface Externos (EIFs)	4	×	5	7	10	= 20
Contagem total						50

“Em vez de ficar apenas meditando sobre “para que serve a nova métrica”... Deveríamos fazer também a pergunta mais básica, “O que vamos fazer com as métricas?”

Michael Mah e Larry Putnam

Com base no valor FP projetado, derivado do modelo de requisitos, a equipe de projeto pode estimar o tamanho total implementado da função de interação de usuário do *CasaSegura*. Vamos assumir que dados já conhecidos indicam que um FP se traduz em 60 linhas de código (será usada uma linguagem orientada a objeto) e que 12 FPs são produzidos por cada pessoa-mês de trabalho. Esses dados históricos fornecem ao gerente do projeto informações importantes de planejamento baseadas no modelo de requisitos e não nas estimativas preliminares. Suponha ainda que projetos anteriores tenham apresentado uma média de três erros por ponto de função durante as revisões de requisitos e projeto e quatro erros por ponto de função durante o teste de unidade e integração. Esses dados podem finalmente ajudá-lo a avaliar a totalidade das suas atividades de revisão e teste.

Uemura e seus colegas [Uem99] sugerem que pontos de função podem também ser computados por meio dos diagramas de classe UML e sequência. Se você tiver mais interesse, veja os detalhes em [Uem99].

23.2.2 Métricas para qualidade de especificação

Davis e seus colegas [Dav93] propõem uma lista de características que podem ser usadas para avaliar a qualidade do modelo de requisitos e as especificações de requisitos correspondentes: *peculiaridade* (ausência de ambiguidade), *totalidade*, *exatidão*, *entendimento*, *repetibilidade*, *consistência interna e externa*, *disponibilidade*, *brevidade*, *rastreabilidade*, *modificação*, *precisão* e *reutilização*. Além disso, os autores observam que há especificações de alta qualidade armazenadas eletronicamente; executáveis ou pelo menos interpretáveis; comentadas de acordo com sua importância relativa; e estáveis versões atualizadas, organizadas referências cruzadas e especificadas com o nível correto de detalhe.

Embora muitas dessas características pareçam ser qualitativas em sua natureza, Davis et al. [Dav93] sugerem que cada uma delas pode ser representada usando uma ou mais métricas. Por exemplo, assumimos que há nr requisitos em uma especificação, tal que

$$n_r = n_f + n_{nf}$$

em que n_f é o número de requisitos funcionais e n_{nf} é o número de requisitos não funcionais (por exemplo, desempenho).

Para determinarem a *peculiaridade* (ausência de ambiguidade) dos requisitos, Davis et al. sugerem uma métrica baseada na consistência da interpretação de cada requisito por parte dos revisores:

$$Q_1 = \frac{n_{ui}}{n_r}$$

PONTO-CHAVE

Medindo-se características da especificação, é possível obter informações quantitativas sobre peculiaridade e totalidade.

em que n_{ui} é o número de requisitos para os quais todos os revisores tiveram interpretações idênticas. Quanto mais próximo o valor Q estiver de 1, mais baixa será a ambiguidade da especificação.

A *totalidade* dos requisitos funcionais pode ser determinada calculando-se a relação

$$Q_2 = \frac{n_u}{n_i \times n_s}$$

em que n_u é o número de requisitos funcionais únicos, n_i é o número de entradas (estímulos) definidas ou implícitas pela especificação e n_s é o número de estados especificados. A relação Q_2 mede a porcentagem de funções necessárias especificadas para um sistema. No entanto, ela não trata de requisitos não funcionais. Para incorporar esses requisitos em uma métrica global para a totalidade, você deve considerar o grau com o qual os requisitos foram validados:

$$Q_3 = \frac{n_c}{n_c + n_{nv}}$$

em que n_c é o número de requisitos validados como corretos e n_{nv} é o número de requisitos que não foram ainda validados.

“Meça aquilo que é mensurável, e o que não for torne mensurável.”

Galileu

23.3 MÉTRICAS PARA O MODELO DE PROJETO

PONTO-CHAVE

As métricas podem proporcionar informações sobre dados estruturais e complexidade do sistema associadas ao projeto da arquitetura.

É inconcebível que o projeto de um novo avião, um novo chip de computador ou um novo edifício comercial seja conduzido sem a definição de medidas, sem determinar métricas para os vários aspectos de qualidade e sem usá-las como indicadores para orientar a maneira pela qual o projeto evoluirá. E, além disso, o projeto de sistemas complexos baseados em software muitas vezes é realizado praticamente sem nenhuma medição. A ironia disso tudo é que as métricas de projeto para software estão disponíveis, mas a grande maioria dos engenheiros de software continua ignorando sua existência.

As métricas de projeto para software de computador, como todas as outras métricas de software, não são perfeitas. Continua o debate sobre sua eficácia e a maneira pela qual deveriam ser aplicadas. Muitos especialistas argumentam que é necessária mais experimentação para que as medições de projeto possam ser usadas. E, além disso, projeto sem medição é uma alternativa inaceitável.

Nas próximas seções, examinaremos algumas das métricas de projeto mais comuns para software de computador. Cada uma delas pode proporcionar uma melhor visualização, e todas podem ajudar o projeto a evoluir para um nível maior de qualidade.

23.3.1 Métricas de projeto da arquitetura

As métricas de projeto da arquitetura focalizam as características da arquitetura do programa (Capítulo 9) com ênfase na estrutura arquitetônica e na eficácia dos módulos ou componentes dentro da arquitetura. Essas métricas são uma “caixa-preta” no sentido de que elas não requerem qualquer conhecimento do funcionamento interno de um determinado componente de software.

Card e Glass [Car90] definem três medidas de complexidade de projeto de software: complexidade estrutural, complexidade de dados e complexidade de sistema.

Para arquiteturas hierárquicas (por exemplo, arquiteturas de chamada e retorno), a *complexidade estrutural* de um módulo i é definida da seguinte maneira:

$$S(i) = f_{out}^{(i)2}$$

em que $f_{out}^{(i)}$ é o fan-out⁶ do módulo i .

6 Fan-out é definido como o número de módulos imediatamente subordinados ao módulo i ; o número de módulos chamados diretamente pelo módulo i .

A *complexidade dos dados* (*data complexity*) proporciona uma indicação da complexidade na interface interna para um módulo i e é definida como

$$D(i) = \frac{v(i)}{f_{\text{out}}(i) + 1}$$

em que $v(i)$ é o número de variáveis de entrada e saída passadas para e do módulo i .

Por fim, a *complexidade do sistema* (*system complexity*) é definida como a soma da complexidade estrutural e de dados, especificada como

$$C(i) = S(i) + D(i)$$

À medida que esses valores de complexidade aumentam, a complexidade global da arquitetura do sistema também aumenta. Isso leva a uma maior probabilidade de que o trabalho de integração e teste também aumente.

Fenton [Fen91] sugere um conjunto de métricas de morfologia simples (isto é, forma) que permite que diferentes arquiteturas de programa sejam comparadas usando uma série de dimensões diretas. Referindo-se à arquitetura de chamada e retorno na Figura 23.4, podem ser definidas as seguintes métricas:

$$\text{Tamanho} = n + a$$

em que n é o número de nós e a é o número de arcos. Para a arquitetura mostrada na Figura 23.4,

$$\text{Tamanho} = 17 + 18 = 35$$

Profundidade = caminho mais longo desde o nó raiz (topo) até o nó da folha. Para a arquitetura mostrada na Figura 23.4, profundidade = 4.

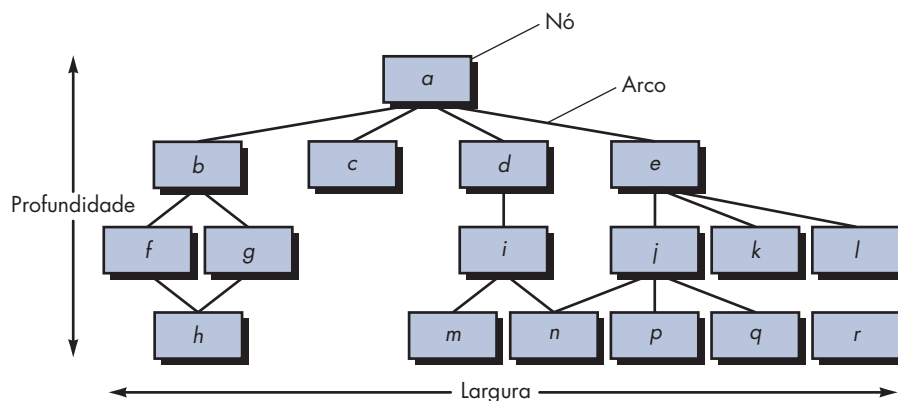
Largura = número máximo de nós em qualquer nível da arquitetura. Para a arquitetura mostrada na Figura 23.4, largura = 6.

A relação arco para nó, $r = a/n$, mede a densidade de conectividade da arquitetura e pode fornecer uma indicação simples do acoplamento da arquitetura. Para a arquitetura mostrada na Figura 23.4, $r = 18 / 17 = 1,06$.

A força aérea americana (U.S. Air Force Systems Command) [USA87] desenvolveu um conjunto de indicadores de qualidade de software baseados nas características de projeto mensuráveis de um programa de computador. Usando conceitos similares àqueles propostos na norma IEEE Std. 982.1-1988 [IEE94], a Força Aérea usa informações obtidas do projeto de dados e de

FIGURA 23.4

Métricas de morfologia



arquitetura para criar um *índice de qualidade de projeto de estrutura* (*design structure quality index* – DSQI) que varia de 0 a 1. Os seguintes valores devem ser apurados para computar o DSQI [Cha89]:

- S_1 = número total de módulos definidos na arquitetura do programa
- S_2 = número de módulos cuja função correta depende da origem dos dados de entrada ou que produzem dados para ser usados em outro lugar (em geral, módulos de controle, entre outros, não seriam contados como parte de S_2)
- S_3 = número de módulos cuja função correta depende de processamento anterior
- S_4 = número de itens de base de dados (inclui objetos dados e todos os atributos que definem objetos)
- S_5 = número total de itens únicos de base de dados
- S_6 = número de segmentos de base de dados (diferentes registros ou objetos individuais)
- S_7 = número de módulos com uma única entrada e saída (processamento de exceção não é considerado múltipla saída)

“A medição pode ser vista como um desvio. Esse desvio é necessário porque os humanos, na sua maioria, não são capazes de tomar decisões claras e objetivas [sem um suporte quantitativo].”

Horst Zuse

Uma vez determinados os valores S_1 até S_7 para um programa de computador, podem ser calculados os seguintes valores intermediários:

Estrutura de programa: D_1 , em que D_1 é definido da seguinte maneira: Se o projeto da arquitetura foi desenvolvido por meio de um método distinto (por exemplo, projeto orientado a fluxo de dados ou projeto orientado a objeto), então $D_1 = 1$, caso contrário $D_1 = 0$.

Independência de módulo: $D_2 = 1 - \frac{S_2}{S_1}$

Módulos não dependentes de processamento anterior: $D_3 = 1 - \frac{S_3}{S_1}$

Tamanho da base de dados: $D_4 = 1 - \frac{S_5}{S_4}$

Divisão da base de dados: $D_5 = 1 - \frac{S_6}{S_4}$

Característica de entrada/saída do módulo: $D_6 = 1 - \frac{S_7}{S_1}$

Com os valores intermediários determinados, o DSQI é computado da seguinte maneira:

$$DSQI = \sum w_i D_i$$

em que $i = 1$ a 6, w_i é o peso relativo da importância de cada um dos valores intermediários e $\sum w_i = 1$ (se todos os D_i tiverem o mesmo peso, então $w_i = 0,167$).

O valor de DSQI para projetos passados pode ser determinado e comparado com um projeto que esteja atualmente em desenvolvimento. Se o DSQI for significativamente mais baixo do que a média, é aconselhável executar mais trabalho de projeto e revisão. De modo semelhante, se grandes alterações deverão ser feitas em um projeto, o efeito daquelas alterações sobre o DSQI pode ser calculado.

23.3.2 Métricas para projeto orientado a objeto

Há muita coisa sobre projeto orientado a objeto que é subjetivo – um projetista experiente “sabe” como caracterizar um sistema orientado a objeto de maneira que implemente efetivamente os requisitos do cliente. Mas, à medida que um modelo de projeto orientado a objeto cresce em tamanho e complexidade, uma visão mais objetiva das características do projeto pode favorecer tanto o projetista experiente (que obtém uma visão adicional) quanto o novato (que obtém uma indicação da qualidade que de outra forma não estaria disponível).

Em um tratamento detalhado de métricas de software para sistemas orientados a objeto, Whitmire [Whi97] descreve nove características distintas e mensuráveis de um projeto orientado a objeto:

 **Que características podem ser medidas quando avaliamos um projeto orientado a objeto?**

“Muitas das decisões para as quais eu tinha que me basear no folclore e em mitos podem agora ser tomadas por meio de dados quantitativos.”

Scott Whitmire

Tamanho. É definido em termos de quatro visualizações: população, volume, comprimento e funcionalidade. *População* é medida tomando-se uma contagem estática das entidades orientadas a objeto, como classes ou operações. Medidas de *volume* são idênticas às medidas de população, mas são coletadas dinamicamente – em determinado instante do tempo. *Comprimento* é a medida de uma cadeia de elementos de projeto interconectados (por exemplo, a extensão de uma árvore de herança é uma medida do comprimento). As métricas de *funcionalidade* proporcionam uma indicação indireta do valor fornecido ao cliente por uma aplicação orientada a objeto.

Complexidade. Assim como o tamanho, há muitas visões diferentes da complexidade do software [Zus97]. Whitmire vê a complexidade em termos de características estruturais examinando como as classes de um projeto orientado a objeto se inter-relacionam umas com as outras.

Acoplamento. As conexões físicas entre elementos do projeto orientado a objeto (por exemplo, o número de colaborações entre classes ou o número de mensagens passadas entre objetos) representam o acoplamento em um sistema orientado a objeto.

Suficiência. Whitmire define *suficiência* como “o grau com o qual uma abstração possui as características requeridas para ela, ou o grau com o qual um componente de projeto possui características em sua abstração, do ponto de vista da aplicação corrente”. Em outras palavras, perguntamos: “Que propriedades essa abstração (classe) precisa ter para ser útil para mim?” [Whi97]. Essencialmente, um componente de projeto (por exemplo, uma classe) é *suficiente* se refletir totalmente todas as propriedades do objeto de domínio da aplicação que está modelando – isto é, que a abstração (classe) possui as características necessárias para ele.

Totalidade. A única diferença entre totalidade e suficiência é “o conjunto de características em relação às quais comparamos a abstração ou o componente de projeto” [Whi97]. A *suficiência* compara a abstração do ponto de vista da aplicação corrente. A *totalidade* considera múltiplos pontos de vista, formulando a pergunta: “Que propriedades são necessárias para representar completamente o objeto de domínio do problema?”. Em virtude do critério da totalidade considerar diferentes pontos de vista, ele tem uma implicação indireta no grau com o qual a abstração ou o componente de projeto pode ser reutilizado.

Coesão. Assim como seu correspondente no software convencional, um componente orientado a objeto deverá ser projetado de maneira que tenha todas as operações funcionando em conjunto para atingir um objetivo único e bem definido. A coerência de uma classe é determinada examinando-se o grau segundo o qual “o conjunto de propriedades que ela possui faz parte do domínio do problema ou projeto” [Whi97].

Originalidade. Uma característica similar à simplicidade, originalidade (aplicada tanto a operações quanto a classes) é o grau segundo o qual uma operação é atômica – isto é, a operação não pode ser construída por meio de uma sequência de outras operações contidas dentro de uma classe. Uma classe que apresenta um alto grau de originalidade encapsula apenas operações primitivas.

Similaridade. O grau segundo o qual duas ou mais classes são similares em termos de sua estrutura, função, comportamento ou finalidade é indicado por essa medição.

Volatilidade. Conforme já afirmamos muitas vezes neste livro, podem ocorrer alterações de projeto quando os requisitos são modificados ou quando acontecem modificações em outras partes de um aplicativo, resultando em adaptação obrigatória do componente de projeto em questão. A volatilidade de um componente orientado a objeto mede a possibilidade de que uma alteração venha a ocorrer.

Na realidade, as métricas de produto para sistemas orientados a objeto podem ser aplicadas não só ao modelo de projeto, mas também ao modelo de requisitos. Nas próximas seções, discutiremos as métricas que fornecem indicação da qualidade em nível de classe orientada a

objeto e em nível de operação. Além disso, serão exploradas também métricas aplicáveis a gerenciamento de projeto e teste.

23.3.3 Métricas orientadas a classe — o conjunto de métricas CK

A classe é a unidade fundamental de um sistema orientado a objeto. Portanto, medidas e métricas para uma classe individual, para a hierarquia da classe e colaborações entre classes serão valiosas quando tivermos de avaliar qualidade de projeto orientado a objeto. Uma classe encapsula dados e a função manipula os dados. Com frequência é o “pai” das subclasses (às vezes chamadas de filhas) que herda seus atributos e operações. Muitas vezes elas colaboram com outras classes. Cada uma dessas características pode ser usada como base para a medida.⁷

Chidamber e Kemerer propuseram um dos conjuntos mais amplamente conhecidos de métricas de software orientado a objeto [Chi94]. Também chamado de conjunto de métricas CK (*CK metrics suite*), os autores propuseram seis métricas de projeto baseado em classe para sistemas orientados a objeto.⁸

Métodos ponderados por classe (*weighted methods per class – WMC*). Suponha que n métodos de complexidade $c_1, c_2, \dots, c_n$ são definidos para uma classe C . A métrica específica de complexidade escolhida (por exemplo, complexidade ciclomática) deverá ser normalizada de maneira que a complexidade nominal para um método assuma o valor 1.0.

$$WMC = \sum c_i$$

para $i = 1$ a n . O número de métodos e sua complexidade são indicadores razoáveis do trabalho necessário para implementar e testar uma classe. Além disso, quanto maior for o número de métodos, mais complexa será a árvore de herança (todas as subclasses herdam os métodos de seus pais). Por fim, conforme o número de métodos cresce para uma dada classe, ela tende a se tornar cada vez mais específica de aplicação, limitando assim sua potencial reutilização. Por todas essas razões, o WMC deverá ser mantido o mais baixo possível.

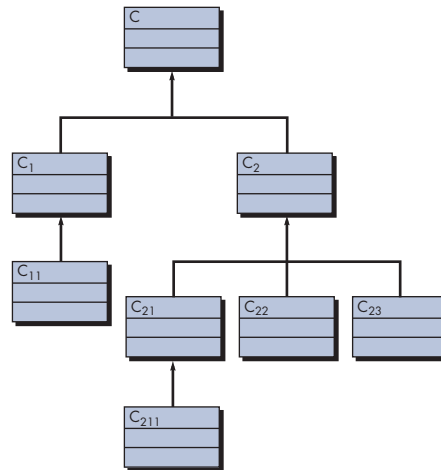
Embora pudesse parecer relativamente fácil desenvolver uma contagem para o número de métodos em uma classe, o problema é na realidade mais complexo do que parece. Deverá ser desenvolvida uma abordagem consistente de contagem [Chu95].

Extensão da árvore de herança (*depth of the inheritance tree – DIT*). Essa métrica é “o comprimento máximo desde o nó até a raiz da árvore” [Chi94]. De acordo com a Figura 23.5, o valor da DIT para a hierarquia de classe mostrada é 4. À medida que a DIT cresce, é possível que classes de nível inferior herdem muitos métodos. Isso causa dificuldades potenciais quando se tenta prever o comportamento de uma classe. Uma hierarquia de classe profunda (com DIT grande) também leva a uma complexidade maior no projeto. No lado positivo, grandes valores DIT implicam que muitos métodos podem ser reutilizados.

Número de filhas (*number of children – NOC*). As subclasses que estão imediatamente subordinadas a uma classe na hierarquia de classe são chamadas de suas filhas. De acordo com a Figura 23.5, a classe C_2 tem três filhas – as subclasses C_{21} , C_{22} e C_{23} . Conforme cresce o número de filhas, a reutilização aumenta, mas também à medida que o NOC aumenta, a abstração representada pela classe pai pode ser diluída se algumas das filhas não forem membros apropriados da classe pai. Conforme o NOC aumenta, a quantidade de teste (necessário para exercitar cada filha em seu contexto operacional) também aumentará.

7 Deve-se notar que a validade de algumas das métricas abordadas neste capítulo são atualmente discutidas na literatura técnica. Aqueles que defendem a teoria de medição demandam um grau de formalismo que algumas métricas orientadas a objeto não fornecem. No entanto, é razoável afirmar que as métricas citadas proporcionam informações úteis para o engenheiro de software.

8 Chidamber, Darcy e Kemerer usam o termo *métodos* em vez de *operações*. O uso desse termo é citado nesta seção.

FIGURA 23.5**Uma hierarquia de classe**

Os conceitos de acoplamento e coesão se aplicam ao software convencional e ao software orientado a objeto. Mantenha o acoplamento de classes baixo e a coesão de operação alta.

Acoplamento entre objetos de classes (coupling between object classes – CBO). O modelo CRC (Capítulo 6) pode ser usado para determinar o valor para o CBO. Essencialmente, CBO é o número de colaborações listadas para uma classe em seu cartão de indexação CRC.⁹ À medida que o CBO aumenta, é possível que a reutilização de uma classe diminua. Altos valores de CBO também complicam modificações e o teste resultante dessas modificações. Em geral, os valores de CBO para cada classe deverão ser mantidos o mais baixos possível; é consistente com a diretriz geral de reduzir o acoplamento em software convencional.

Resposta para uma classe (response for a class – RFC). O conjunto de respostas de uma classe é “um conjunto de métodos que podem potencialmente ser executados em resposta a uma mensagem recebida por um objeto daquela classe” [Chi94]. RFC é o número de métodos no conjunto de respostas. Conforme a RFC aumenta, o trabalho necessário para o teste também aumenta porque a sequência de testes (Capítulo 19) cresce. Segue também que, à medida que a RFC aumenta, a complexidade geral de projeto da classe aumenta.

Falta de coesão em métodos (lack of cohesion in methods – LCOM). Cada método dentro de uma classe **C** acessa um ou mais atributos (também chamados de variáveis de instância). LCOM é o número de métodos que acessam um ou mais dos mesmos atributos.¹⁰ Se nenhum método acessa os mesmos atributos, LCOM = 0. Para ilustrar o caso em que LCOM ≠ 0, considere uma classe com seis métodos. Quatro dos métodos têm um ou mais atributos em comum (eles acessam atributos comuns). Portanto, LCOM = 4. Se LCOM for alto, métodos podem ser acoplados uns aos outros via atributos. Isso aumenta a complexidade do projeto de classe. Embora haja casos em que um valor alto de LCOM seja justificável, é desejável manter a coesão alta; isto é, manter LCOM baixo.¹¹

⁹ Se os cartões de indexação CRC são desenvolvidos manualmente, a totalidade e a consistência devem ser avaliadas para que o CBO possa ser determinado de forma confiável.

¹⁰ A definição formal é um pouco mais complexa. Veja os detalhes em [Chi94].

¹¹ A métrica LCOM fornece informações úteis em algumas situações, mas pode ser confusa em outras. Por exemplo, mantendo-se o acoplamento encapsulado (dentro de uma classe), aumenta a coesão do sistema como um todo. Portanto, pelo menos em um sentido importante, LCOM mais alto realmente sugere que uma classe possa ter coesão mais alta, não mais baixa.

CASA SEGURA



Aplicando métricas CK

Cena: Sala do Vinod.

Atores: Vinod, Jamie, Shakira e Ed – membros da equipe de engenharia de software do CasaSegura que continuam a trabalhar no projeto em nível de componente e projeto de caso de teste.

Conversa:

Vinod: Vocês tiveram uma chance de ler a descrição sobre o conjunto de métricas CK que eu enviei na quarta-feira e fazer aquelas medidas?

Shakira: Não foi muito complicado. Utilizei minha classe UML e diagramas de sequência, como você sugeriu, e fiz contagens aproximadas de DIT, RFC e LCOM. Não consegui encontrar o modelo CRC, então não fiz contagem de CBO.

Jamie (rindo): Você não poderia encontrar o modelo CRC porque ele estava comigo.

Shakira: É isso que eu adoro nesta equipe, uma excelente comunicação.

Vinod: Fiz minhas contagens... Vocês desenvolveram valores para as métricas CK? [Jamie e Ed acenam afirmativamente.]

Jamie: Como eu tinha os cartões CRC, dei uma olhada no CBO e ele parecia bastante uniforme na maioria das classes. Havia apenas uma exceção.

Ed: Há algumas classes em que o RFC é muito alto, comparado com as médias... Talvez devêssemos dar um jeito de simplificá-los.

Jamie: Talvez sim, talvez não. Ainda estou preocupado com o prazo, e não quero corrigir coisas que não tenham realmente erros.

Vinod: Concorro com isso. Talvez vocês devessem dar uma olhada nas classes que apresentam números ruins em pelo menos duas ou mais das métricas CK.

Shakira (observando a lista de classes de Ed com alto RFC): Olhe, veja esta classe, com LCOM e RFC ambos altos.

Vinod: Penso que sim... Será difícil de implementar devido à complexidade e difícil de testar pela mesma razão. Provavelmente vale a pena projetar duas classes separadas para conseguir o mesmo comportamento.

Jamie: Você acha que modificando economizaremos tempo?

Vinod: No longo prazo, sim.

“Analisar o software orientado a objeto para avaliar sua qualidade está se tornando cada vez mais importante na medida em que o paradigma orientado a objeto continua a crescer em popularidade.”

Rachel Harrison
et al.

23.3.4 Métricas orientadas a classe – o conjunto de métricas MOOD

Harrison, Counsell e Nithi [Har98b] propõem um conjunto de métricas para projeto orientado a objeto que proporciona indicadores quantitativos para características de projeto orientado a objeto. Veja a seguir exemplos das métricas MOOD.

Fator de herança de método (method inheritance factor – MIF). O grau segundo o qual a arquitetura de classe de um sistema orientado a objeto faz uso de herança tanto para métodos (operações) quanto para atributos. É definido como

$$MIF = \frac{\sum M_i(C_i)}{\sum M_a(C_i)}$$

em que a somatória ocorre com $i = 1$ até TC. TC é definido como o número total de classes na arquitetura, C_i é a classe dentro da arquitetura e

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

em que

$M_d(C_i)$ = número de métodos que podem ser invocados em associação com C_i

$M_a(C_i)$ = número de métodos declarados na classe C_i

$M_i(C_i)$ = número de métodos herdados (e não cancelados) em C_i

O valor de MIF [o fator de herança de atributo (AIF) é definido de maneira análoga] fornece uma indicação do impacto da herança sobre o software orientado a objeto.

Fator de acoplamento (coupling factor – CF). Anteriormente, neste capítulo, observou-se que o acoplamento é uma indicação das conexões entre elementos do projeto orientado a objeto. O conjunto de métricas MOOD define o acoplamento da seguinte maneira:

$$CF = \sum_i \sum_j is_client \frac{(C_i, C_j)}{T_c^2 - T_c}$$

em que a somatória de $i = 1$ a T_c e $j = 1$ a T_c . A função

$$is_client = 1, \text{ se, e somente se, existir uma relação entre a classe-cliente } C_c \text{ e a classe-servidora } C_s, \text{ e } C_c \neq C_s \\ = 0, \text{ em caso contrário}$$

Embora muitos fatores afetem a complexidade, entendimento e sustentabilidade do software, é razoável concluir que, conforme o fator CF aumenta, a complexidade do software orientado a objeto também aumenta e o entendimento, a sustentabilidade e o potencial de reutilização podem ficar prejudicados.

Harrison e seus colegas [Har98b] apresentam uma análise detalhada do MIF e CF com suas métricas e examinam sua validade para uso na avaliação da qualidade do projeto.

23.3.5 Métricas orientadas a objeto propostas por Lorenz e Kidd

Em seu livro sobre métricas orientadas a objeto, Lorenz e Kidd [Lor94] dividem as métricas baseadas em classe em quatro categorias principais, cada uma delas ocupando uma posição no projeto em nível de componente: tamanho, herança, aspectos internos e externos. As métricas orientadas a tamanho para uma classe de projeto orientado a objeto focalizam contagens de atributos e operações para uma classe individual e valores médios para o sistema orientado a objeto como um todo. Métricas baseadas em herança focalizam a maneira pela qual as operações são reutilizadas através da hierarquia de classe. Métricas para aspectos internos de classe procuram a coesão (Seção 23.3.3) e questões orientadas a código, e as métricas de aspectos externos examinam acoplamento e reutilização. Um exemplo das métricas propostas por Lorenz e Kidd é:

Tamanho de classe (class size – CS). O tamanho global de uma classe pode ser determinado por meio das seguintes medidas:

- número total de operações (operações de instâncias herdadas e privadas) que são encapsuladas dentro da classe
- número de atributos (atributos de instâncias herdadas e privadas) que são encapsulados pela classe

A métrica WMC proposta por Chidamber e Kemerer (Seção 23.3.3) é também uma medida ponderada do tamanho da classe. Conforme observamos, valores altos de CS indicam que uma classe pode ter muita responsabilidade. Isso reduzirá a reutilização da classe e complicará a implementação e o teste. Em geral, operações e atributos, herdados ou públicos, deverão ser ponderados mais intensamente na determinação do tamanho da classe [Lor94]. Operações privadas e atributos privados permitem especialização e são mais localizados no projeto. Podem ser calculados também valores médios para o número de atributos e operações de classe. Quanto mais baixos os valores médios para o tamanho, maior probabilidade terão as classes dentro do sistema de ser reutilizadas mais amplamente.

23.3.6 Métricas de projeto em nível de componente

As métricas de projeto em nível de componente para componentes convencionais de software focalizam características internas de um componente de software e incluem medidas dos “três Cs” – coesão de módulo (*module cohesion*), acoplamento (*coupling*) e complexidade (*complexity*). Essas medidas podem ajudá-lo a avaliar a qualidade de um projeto em nível de componente.



Durante a revisão do modelo de análise, os cartões de indexação CRC fornecerão uma indicação razoável dos valores esperados para CS. Se você encontrar uma classe com um alto número de responsabilidades, pense em dividi-la.

PONTO-CHAVE

É possível computar medidas da independência funcional – acoplamento e coesão – de um componente e usá-las para avaliar a qualidade de um projeto.

Métricas de projeto em nível de componente podem ser aplicadas uma vez desenvolvido o projeto procedural; elas são uma “caixa de vidro” pelo fato de requererem conhecimento do funcionamento interno do módulo que está sendo considerado. Como alternativa, elas podem ser adiadas até que o código-fonte esteja disponível.

Métricas de coesão. Definidas por Bieman e Ott [Bie94] como uma coleção de métricas que fornecem a indicação da coerência (Capítulo 8) de um módulo. As métricas são caracterizadas em termos de cinco conceitos e medidas:

- *Fatia de dados (data slice).* Em poucas palavras, é a busca retroativa através de um módulo à procura por valores de dados que afetam a localização do módulo na qual essa busca começou. Devemos observar que podem ser definidas tanto as fatias de programa (que focalizam instruções e condições) como as fatias de dados.
- *Fichas de dados (data tokens).* As variáveis definidas para um módulo podem ser consideradas fichas de dados para o módulo.
- *Fichas aglutinantes (glue tokens).* Conjunto de fichas de dados que residem em uma ou mais fatias de dados.
- *Fichas superaglutinantes (superglue tokens).* Fichas de dados comuns a todas as fatias de dados em um módulo.
- *Aglutinação (stickiness).* A aglutinação relativa de uma ficha aglutinante é diretamente proporcional ao número de fatias de dados que ele une.

Bieman e Ott desenvolvem métricas para *coesão funcional forte (strong functional cohesion - SFC)*, *coesão funcional fraca (weak functional cohesion - WFC)* e adesividade (*adhesiveness*) - o grau relativo em que fichas aglutinantes juntam fatias de dados entre si. Para uma discussão detalhada das métricas de Bieman e Ott, é aconselhável consultar os próprios autores [Bie94].

Métricas de acoplamento (coupling metrics). O acoplamento de módulo fornece uma indicação da “conexão” de um módulo com outros módulos, dados globais e o ambiente externo. No Capítulo 9, o acoplamento foi discutido em termos qualitativos.

Dhama [Dha95] propôs uma métrica para acoplamento de módulo, que abrange acoplamento de dados, de fluxo de controle, global e ambiental. As medidas necessárias para computar o acoplamento de módulo são definidas em relação a cada um dos três tipos de acoplamento citados anteriormente.

Para acoplamento de dados e de fluxo de controle,

$$\begin{aligned}d_i &= \text{número de parâmetros de dados de entrada} \\c_i &= \text{número de parâmetros de controle de entrada} \\d_o &= \text{número de parâmetros de dados de saída} \\c_o &= \text{número de parâmetros de controle de saída}\end{aligned}$$

Para acoplamento global,

$$\begin{aligned}g_d &= \text{número de variáveis globais usadas como dados} \\g_c &= \text{número de variáveis globais usadas como controle}\end{aligned}$$

Para acoplamento ambiental,

$$\begin{aligned}w &= \text{número de módulos chamados (fan-out)} \\r &= \text{número de módulos que chamam o módulo em consideração (fan-in)}\end{aligned}$$

Usando essas medidas, um indicador de acoplamento de módulo m_c é definido da seguinte maneira:

$$m_c = \frac{k}{M}$$

em que k é uma constante de proporcionalidade e

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r$$

Valores para k , a , b e c devem ser obtidos empiricamente.

À medida que o valor para mc aumenta, o acoplamento global do módulo diminui. Para que a métrica de acoplamento suba quando o grau de acoplamento aumenta, uma métrica de acoplamento revisada pode ser definida como

$$C = 1 - m_c$$

em que o grau de acoplamento aumenta quando o valor de M aumenta.

Métricas de complexidade (complexity metrics). Uma variedade de métricas de software pode ser computada para determinar a complexidade do fluxo de controle do programa. Muitas dessas métricas são baseadas no diagrama de fluxo. Um gráfico (Capítulo 18) é uma representação composta de nós e vínculos (também chamados de *edges*). Quando os vínculos estão direcionados, o diagrama de fluxo é um grafo direcionado.

McCabe e Watson [McC94] identificam uma série de usos importantes para as métricas de complexidade:

Métricas de complexidade podem ser usadas para prever informações críticas sobre a confiabilidade e sustentabilidade de sistemas de software com base em análise automática do código-fonte [ou informações sobre o projeto procedural]. Métricas de complexidade também proporcionam informações durante o projeto de software para ajudar a controlar a [atividade de projeto]. Durante os testes e a manutenção, elas fornecem informações detalhadas sobre os módulos de software para ajudar a apontar áreas de instabilidade potencial.

A métrica de complexidade mais usada (e debatida) para software é a ciclomática, desenvolvida por Thomas McCabe [McC76] e discutida em detalhes no Capítulo 18.

Zuse ([Zus90], [Zus97]) apresenta uma discussão enciclopédica de mais de 18 diferentes categorias de métricas de complexidade de software. O autor aborda as definições básicas para métricas em cada categoria (por exemplo, há um conjunto de variações na métrica de complexidade ciclomática) e então analisa e critica cada uma delas. O trabalho de Zuse é o mais abrangente publicado até agora.

23.3.7 Métricas orientadas a operação

Devido à classe ser a unidade dominante nos sistemas orientados a objeto, poucas métricas têm sido propostas para operações que residem dentro de uma classe. Churcher e Shepperd [Chu95] discutem isso quando afirmam: “Resultados de estudos recentes indicam que métodos tendem a ser pequenos, tanto em termos de número de instruções quanto em complexidade lógica [Wil93], sugerindo que a estrutura de conectividade de um sistema pode ser mais importante do que o conteúdo de módulos individuais”. No entanto, podem ser obtidas algumas informações examinando-se as características médias dos métodos (operações). Três métricas simples, propostas por Lorenz e Kidd [Lor94], são apropriadas:

Tamanho médio de operação (average operation size – OS_{avg}). O tamanho pode ser determinado contando-se o número de linhas de código ou o número de mensagens enviadas pela operação. Na medida em que aumenta o número de mensagens enviadas por uma única operação, é possível que as responsabilidades não tenham sido bem alocadas dentro da classe.

Complexidade de operação (operation complexity – OC). A complexidade de uma operação pode ser calculada usando-se quaisquer métricas de complexidade propostas para software convencional [Zus90]. Como as operações deverão ser limitadas a uma responsabilidade específica, o projetista deve se esforçar para manter a OC o mais baixo possível.

PONTO-CHAVE

A complexidade ciclomática é apenas uma dentre inúmeras métricas de complexidade.

Número médio de parâmetros por operação (*average number of parameters per operation* – NP_{avg}). Quanto maior o número de parâmetros de operação, mais complexa é a colaboração entre objetos. Em geral, NP_{avg} deverá ser mantida o mais baixo possível.

23.3.8 Métricas de projeto de interface de usuário

“Você pode aprender pelo menos um princípio de projeto de interface de usuário quando utiliza uma lavadora de louças. Se você colocar muitos objetos lá dentro, nada ficará muito limpo.”

Autor desconhecido

Embora haja uma literatura significativa sobre o projeto de interface humanos/computadores (Capítulo 11), têm sido publicadas poucas informações sobre métricas que poderiam fornecer informações sobre a qualidade e utilização da interface.

Sears [Sea93] sugere que a *adequação do layout* (*layout appropriateness* LA) é uma métrica de projeto valiosa para interfaces humanos/computadores. Uma GUI típica usa entidades de layout – ícones gráficos, texto, menus, janelas e itens do gênero – para ajudar o usuário na execução das tarefas. Para executar uma tarefa usando uma GUI, o usuário deve passar de uma entidade de layout para a próxima. A posição absoluta e relativa de cada entidade de layout, a frequência com a qual ela é usada e o “custo” da transição de uma entidade de layout para a próxima contribuem para a adequação da interface.

Um estudo das métricas para páginas da Web [Ivo01] indica que características simples dos elementos do layout podem também ter um impacto significativo sobre a qualidade aparente do projeto da GUI. O número de palavras, links, gráficos, cores e fontes (entre outras características) contidas em uma página da Web afetam a complexidade aparente e a qualidade dela.

É importante notar que a seleção de um projeto de GUI pode ser guiada com métricas como LA, mas o julgamento final deverá ser a entrada de usuário baseada em protótipos de GUI. Nielsen e Levy [Nie94] relatam que “há uma chance razoavelmente grande de sucesso se alguém escolher entre interface [projetos] baseada unicamente nas opiniões dos usuários. O desempenho médio da tarefa do usuário e sua satisfação subjetiva com uma GUI estão altamente correlacionados.”

23.4 MÉTRICAS DE PROJETO PARA WEBAPPS

Um bom conjunto de medidas e métricas para WebApps fornece respostas quantitativas para as seguintes questões:

- A interface de usuário promove a utilização?
- O aspecto estético da WebApp é apropriado para o domínio de aplicação e é agradável ao usuário?
- O conteúdo é projetado de forma a reunir o máximo de informações com o mínimo esforço?
- A navegação é eficiente e direta?
- A arquitetura da WebApp foi projetada para acomodar as metas e objetivos especiais de seus usuários, a estrutura de conteúdo e funcionalidade e o fluxo de navegação requerido para usar o sistema eficientemente?
- Os componentes são projetados de maneira que reduzam a complexidade de procedimento e melhorem a exatidão, confiabilidade e desempenho?



Muitas dessas métricas são aplicáveis a todas as interfaces de usuário e devem ser consideradas em conjunto com as apresentadas na Seção 23.3.8.

Atualmente, cada uma dessas questões pode ser resolvida apenas qualitativamente porque não existe um conjunto qualificado de métricas que forneceriam respostas quantitativas.

Nos próximos parágrafos, apresentamos uma amostra representativa das métricas de projeto para WebApp propostas na literatura. É importante notar que muitas dessas métricas ainda não foram validadas e deverão ser usadas com muito critério.

Métricas de interface. Para WebApps, podem ser consideradas as seguintes medidas de interface:

<i>Métrica sugerida</i>	<i>Descrição</i>
Adequação do layout	Veja a Seção 23.3.8.
Complexidade de layout	Número de regiões distintas ¹² definidas para uma interface
Complexidade da região de layout	Número médio de links distintos por região
Complexidade de reconhecimento	Número médio de itens distintos que o usuário deve examinar antes de tomar uma decisão de navegação ou de entrada de dados
Tempo de reconhecimento	Tempo médio (em segundos) que o usuário gasta para selecionar a ação apropriada para uma tarefa
Trabalho de digitação	Número médio de toques necessários para uma função específica
Esforço de click do mouse	Número médio de cliques do mouse por função
Complexidade de seleção	Número médio de links que podem ser selecionados por página
Tempo de aquisição de conteúdo	Número médio de palavras de texto por página da Web
Carga de memória	Número médio de itens de dados distintos que o usuário deve lembrar para atingir um objetivo específico

Métricas de estética (design gráfico). Por sua natureza, o projeto estético depende da avaliação qualitativa e não é geralmente favorável à medição e às métricas. No entanto, Ivory e seus colegas [Ivo01] propõem um conjunto de medidas que podem ser úteis para avaliar o impacto do projeto estético:

<i>Métrica Sugerida</i>	<i>Descrição</i>
Número de palavras	Número total de palavras que aparecem em uma página
Porcentagem de texto de corpo	Porcentagem de palavras que são texto de corpo <i>versus</i> texto de título (cabecinhos)
Texto de corpo destacado %	Parte do texto de corpo que é destacado (por exemplo, negrito, caixa alta)
Contagem de posicionamento de texto	Mudanças na posição do texto a partir do alinhamento à esquerda
Contagem de grupos de texto	Áreas de texto destacadas com cores, regiões com bordas, réguas ou listas
Contagem de link	Total de links em uma página
Tamanho de página	Total de bytes na página, bem como elementos, gráficos e folhas de estilo
Porcentagem gráfica	Porcentagem de bytes da página que são usados para gráficos (figuras)
Contagem gráfica	Total de figuras na página (não incluindo figuras especificadas em scripts, applets e objetos)
Contagem de cores	Total de cores usadas
Contagem de fonte	Total de fontes empregadas (face + size + bold + italic)

Métricas de conteúdo. Métricas nessa categoria focalizam a complexidade de conteúdo e clusters de objetos conteúdo que são organizados em páginas [Men01].

¹² Uma região distinta é uma área do layout de tela que executa um conjunto específico de funções relacionadas (por exemplo, uma barra de menu, uma tela gráfica estática, uma área de conteúdo, uma tela animada).

<i>Métrica Sugerida</i>	<i>Descrição</i>
Espera de página	Tempo médio necessário para que uma página seja baixada em diferentes velocidades de conexão
Complexidade da página	Número médio de diferentes tipos de mídia usada na página, não incluindo texto
Complexidade gráfica	Número médio de mídia gráfica por página
Complexidade de áudio	Número médio de mídias de áudio por página
Complexidade de vídeo	Número médio de mídia de vídeo por página.
Complexidade de animação	Número médio de animações por página
Complexidade de imagem escaneada	Número médio de imagens escaneadas por página

Métricas de navegação. Métricas nessa categoria tratam da complexidade do fluxo de navegação [Men01]. Em geral, são direcionadas apenas para aplicações Web estáticas, que não incluem links e páginas gerados dinamicamente.

<i>Métrica Sugerida</i>	<i>Descrição</i>
Complexidade de link de página	Número de links por página
Conectividade	Número total de links internos, não incluindo links gerados dinamicamente
Densidade de conectividade	Conectividade dividida por número de página

Com um subconjunto das métricas sugeridas, pode ser possível derivar relações empíricas que permitem a uma equipe de desenvolvimento de WebApp avaliar a qualidade técnica e prever o trabalho necessário com base nas estimativas projetadas da complexidade. Há ainda muito a ser feito nesta área.

FERRAMENTAS DO SOFTWARE



Métricas técnicas para WebApps

Objetivo: ajudar os engenheiros da Web a desenvolver métricas para WebApp com significado que forneçam informações sobre a qualidade global de um aplicativo.

Mecânica: o mecanismo das ferramentas é variado.

Ferramentas representativas:¹³ Netmechanic Tools, desenvolvida pela Netmechanic (www.netmechanic.com), é uma coleção de ferramentas que ajudam a melhorar o desempenho do site, focalizando assuntos específicos de implementação. NIST Web Metrics Testbed, desenvolvida pelo National Institute of Standards and Technology (zing.ncsl.nist.gov/WebTools/), abrange a seguinte coleção de ferramentas úteis disponíveis para download:

Web Static Analyzer Tool (WebSAT) – verifica o HTML da página da Web segundo diretrizes de utilização típica.

Web Category Analysis Tool (WebCAT) – permite ao engenheiro de utilização criar e executar uma análise de categoria Web.

Web Variable Instrumenter Program (WebVIP) – capacita um site a capturar um log de interação de usuário.

Framework for Logging Usability Data (FLUD) – implementa um formatador de arquivo e um analisador sintático (*parser*) para representação dos logs de interação do usuário.

VisVIP Tool – produz uma visualização 3D dos caminhos de navegação do usuário através de um site.

TreeDec – acrescenta auxílios de navegação às páginas de um site.

23.5 MÉTRICAS PARA CÓDIGO-FONTE

A teoria de Halstead da “ciência do software” [Hal77] propôs as primeiras “leis” analíticas para programas de computador.¹⁴ Halstead atribuiu leis quantitativas ao desenvolvimento de

¹³ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

¹⁴ Deve-se observar que as “leis” de Halstead geraram controvérsias substanciais, e muitos acreditam que a teoria na qual se baseiam apresenta falhas. No entanto, executaram-se verificações experimentais para algumas linguagens de programação (por exemplo, [Fel89]).

"O cérebro humano segue uma série de regras mais rígidas [para desenvolver algoritmos] do que ele próprio poderia imaginar."

Maurice Halstead

software, usando um conjunto de medidas primitivas que podem ser obtidas depois que o código é gerado, ou estimadas quando o projeto estiver completo. As medidas são:

n_1 = número de operadores distintos que aparecem em um programa

n_2 = número de operandos distintos que aparecem em um programa

N_1 = número total de ocorrências de operador

N_2 = número total de ocorrências de operando

Halstead usa essas medidas primitivas para desenvolver expressões para o tamanho global do programa, volume mínimo potencial para um algoritmo, o volume atual (número de bits necessários para especificar um programa), nível do programa (medida da complexidade do software), nível de linguagem (constante para uma dada linguagem) e outras características como esforço de desenvolvimento, tempo de desenvolvimento e até um número projetado de falhas no software.

Halstead mostra que o tamanho N pode ser estimado da seguinte maneira:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

e o volume do programa pode ser definido como:

$$V = N \log_2 (n_1 + n_2)$$



Operadores incluem todas as construções (constructs) de controle de fluxo, operações condicionais e matemáticas. Operandos abrangem todas as variáveis e constantes de programas.

Devemos notar que V varia com a linguagem de programação e representa o volume de informações (em bits) necessário para especificar um programa.

Teoricamente, deve existir um volume mínimo para determinado algoritmo. Halstead define a razão de volume L como a razão entre o volume da forma mais compacta de um programa e o volume do programa real. Na realidade, L deve ser sempre menor do que 1. Em termos de medidas primitivas, a relação de volume pode ser expressa como

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

O trabalho de Halstead é favorável à verificação experimental e muitas pesquisas já foram feitas para investigar a ciência do software. Uma discussão desse trabalho está além do escopo deste livro. Para mais informações, veja [Zus90], [Fen91] e [Zus97].

23.6 MÉTRICAS PARA TESTE

PONTO-CHAVE

As métricas de teste se classificam em duas grandes categorias: (1) métricas que tentam prever o número provável de testes necessários em vários níveis de teste e (2) métricas que focalizam a abrangência do teste para determinado componente.

Embora muito se tenha escrito sobre métricas de software para teste (por exemplo, [Het93]), a maioria das métricas propostas focaliza o processo de teste, não as características técnicas dos próprios testes. Em geral, os testadores precisam se basear nas métricas de análise, projeto e código para guiá-los no projeto e execução dos casos de teste.

As métricas de projeto da arquitetura fornecem informações sobre a facilidade ou dificuldade associadas com o teste de integração (Seção 23.3) e a necessidade de software de teste especializado (por exemplo, pseudocontroladores (*drivers*) e pseudocontrolados (*stubs*)). A complexidade ciclomática (uma métrica de projeto em nível de componente) depende do teste de caminho básico, um método de projeto de caso de teste apresentado no Capítulo 18. Além disso, a complexidade ciclomática pode ser usada para escolher módulos como candidatos para testes de unidade extensivos. Módulos com alta complexidade ciclomática são mais propensos a apresentar erro do que os de complexidade menor. Por essa razão, você deve dispendar esforços acima da média para descobrir erros nesses módulos antes de serem integrados em um sistema.

23.6.1 Métricas de Halstead aplicadas ao teste

O trabalho de teste pode ser estimado por meio de métricas obtidas das medidas de Halstead (Seção 23.5). Usando as definições para volume V de programa e nível PL , o trabalho Halstead e pode ser calculado como

$$PL = \frac{1}{(n_1/2) \times (N_2/n_2)} \quad (23.2a)$$

$$e = \frac{V}{PL} \quad (23.2b)$$

A porcentagem de trabalho de teste global a ser alocado a um módulo k pode ser estimada usando a seguinte relação:

$$\text{Porcentagem de trabalho de teste } (k) = \frac{e(k)}{\sum e(i)} \quad (23.3)$$

em que $e(k)$ é calculado para o módulo k por meio das Equações (23.2), e a somatória no denominador da Equação (23.3) é a soma do trabalho Halstead por todos os módulos do sistema.

23.6.2 Métricas para teste orientado a objeto

As métricas de projeto orientado a objeto apresentadas na Seção 23.3 fornecem uma indicação da qualidade do projeto. Elas fornecem também uma indicação geral do trabalho de teste necessário para exercitar um sistema orientado a objeto. Binder [Bin94b] sugere um conjunto amplo de métricas de projeto que têm influência direta sobre a “testabilidade” de um sistema orientado a objeto. As métricas consideram aspectos do encapsulamento e herança.

Falta de coesão em métodos (*lack of cohesion in methods – LCOM*).¹⁵ Quanto mais alto o valor de LCOM, mais estados precisam ser testados para garantir que os métodos não gerem efeitos colaterais.

Porcentagem pública e protegida (*percent public and protected – PAP*). Atributos públicos são herdados de outras classes e, portanto, são visíveis àquelas classes. Atributos protegidos são acessíveis a métodos em subclasses. Essa métrica indica a porcentagem de atributos de classe públicos ou protegidos. Altos valores para PAP aumentam a probabilidade de efeitos colaterais entre classes porque atributos públicos e protegidos levam a um alto potencial de acoplamento.¹⁶ Os testes devem ser projetados para garantir que esses efeitos colaterais sejam descobertos.

Acesso público a membros de dados (*public access to data members – PAD*). Essa métrica indica o número de classes (ou métodos) que podem acessar atributos de uma outra classe, uma violação do encapsulamento. Altos valores de PAD levam a efeitos colaterais em potencial entre classes. Os testes devem ser projetados de forma que garantam que esses efeitos colaterais sejam descobertos.

Número de classes-raiz (*number of root classes – NOR*). Essa métrica é uma contagem das hierarquias distintas de classe descritas no modelo de projeto. Devem ser desenvolvidos conjuntos de testes para cada classe-raiz e hierarquia de classe correspondente. À medida que o NOR aumenta, o trabalho de teste também aumenta.



O teste orientado a objeto pode ser muito complexo. As métricas podem ajudá-lo a direcionar os recursos de teste para threads, cenários e pacotes de classes que são “suspeitos” com base nas características medidas. Use-as.

¹⁵ Veja na Seção 23.3.3 uma descrição de LCOM.

¹⁶ Algumas pessoas fazem projetos nos quais nenhum dos atributos é público ou privado, ou seja, $PAP = 0$. Isso implica que todos os atributos devem ser acessados em outras classes via métodos.

Fan-in (FIN). Quando usado no contexto orientado a objeto, fan-in na hierarquia de herança é uma indicação de herança múltipla. $FIN > 1$ indica que uma classe herda seus atributos e operações de mais de uma classe-raiz. $FIN > 1$ deve ser evitado sempre que possível.

Número de filhos (number of children – NOC) e altura da árvore de herança (depth of the inheritance tree – DIT).¹⁷ Conforme mencionamos no Capítulo 19, métodos de superclasse terão de ser retestados para cada subclasse.

23.7 MÉTRICAS PARA MANUTENÇÃO

Todas as métricas de software introduzidas neste capítulo podem ser usadas para o desenvolvimento de novo software e para manutenção do software existente. No entanto, existem também métricas projetadas explicitamente para atividades de manutenção.

A norma IEEE Std. 982.1-1988 [IEE93] sugere um *índice de maturidade de software* (software maturity index – SMI) que fornece uma indicação da estabilidade de um produto (com base nas alterações que ocorrem para cada versão do produto). São determinadas as seguintes informações:

M_T = número de módulos na versão atual

F_c = número de módulos na versão atual que foram alterados

F_a = número de módulos na versão atual que foram acrescentados

F_d = número de módulos da versão anterior que foram excluídos na versão atual

O índice de maturidade de software é calculado da seguinte maneira:

$$SMI = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

À medida que o SMI se aproxima de 1.0, o produto começa a estabilizar. O SMI pode ser usado também como uma métrica para planejamento de atividades de manutenção de software. O tempo médio para produzir uma versão de um software pode ser correlacionado com o SMI, e podem ser desenvolvidos modelos empíricos para o trabalho de manutenção.

FERRAMENTAS DO SOFTWARE



Métricas de produto

Objetivo: ajudar os engenheiros de software no desenvolvimento de métricas com significado que avaliam os produtos gerados durante o projeto de análise e modelagem, geração de código-fonte e teste.

Mecânica: as ferramentas desta categoria abrangem um grande conjunto de métricas e são implementadas como uma aplicação *stand-alone* ou (mais comumente) como funcionalidade que existe dentro das ferramentas para análise e projeto, codificação ou teste. Em muitos casos, a ferramenta métrica analisa uma representação do software (por exemplo, um modelo UML ou código-fonte) e desenvolve uma ou mais métricas como resultado.

Ferramentas representativas:¹⁸ *Krakatau Metrics*, desenvolvida pela Power Software (www.powersoftware.com/products), calcula complexidade, Halstead e métricas relacionadas a C/C++ e Java.

Metrics4C – desenvolvida pela +1 Software Engineering (www.plus-one.com/Metrics4C_fact_sheet.html), calcula uma variedade de métricas de arquitetura, projeto e orientadas a código, bem como métricas orientadas a projeto.

Rational Rose, distribuída pela IBM (www-304.ibm.com/jct03001c/software/awdtools/developer/rose/), é um conjunto de ferramentas para modelagem UML que incorpora uma série de características de análise de métricas.

RSM, desenvolvida pela M-Squared Technologies (msquaredtechnologies.com/m2rsm/index.html), calcula uma grande variedade de métricas orientadas a código para C, C++ e Java.

Understand, desenvolvida pela Scientific Toolworks, Inc. (www.scitools.com), calcula métricas orientadas a código para uma variedade de linguagens de programação.

¹⁷ Veja na Seção 23.3.3 uma descrição de NOC e DIT.

¹⁸ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

23.8 RESUMO

As métricas de software proporcionam uma maneira quantitativa de avaliar a qualidade dos atributos internos do produto, possibilitando avaliar a qualidade antes que ele seja criado. As métricas proporcionam as informações necessárias para criar requisitos eficazes e modelos de projeto, código sólido e testes completos.

Para ser útil na prática, uma métrica de software deve ser simples e computável, persuasiva, consistente e objetiva. Ela deve ser independente da linguagem de programação e fornecer um retorno eficaz.

Métricas para o modelo de requisitos focalizam função, dados e comportamento – os três componentes do modelo. Métricas para projeto consideram aspectos de arquitetura, projeto em nível de componente e projeto de interface. Métricas de projeto da arquitetura consideram os aspectos estruturais do modelo de projeto. Métricas de projeto em nível de componente fornecem indicação da qualidade do módulo, estabelecendo medidas indiretas para coesão, acoplamento e complexidade. Métricas de projeto de interface de usuário fornecem indicação da facilidade com que uma GUI pode ser usada. Métricas para WebApp consideram aspectos da interface de usuário, bem como a estética da WebApp, conteúdo e navegação.

Métricas para sistemas orientados a objeto concentram-se em medidas que podem ser aplicada às características da classe e do projeto – localização, encapsulamento, ocultamento de informações, herança e técnicas de abstração de objeto – que tornam a classe única. O conjunto de métricas CK define seis métricas de software orientado à classe que focalizam a classe e a hierarquia de classe. O conjunto de métricas também desenvolve métricas para avaliar as colaborações entre classes e a coesão de métodos que residem em uma classe. Em um nível orientado a classe, o conjunto de métricas CK pode ser ampliado com métricas propostas por Lorenz e Kidd e o conjunto de métricas MOOD.

Halstead apresenta um conjunto fascinante de métricas em nível de código-fonte. Usando uma série de operadores e operandos presentes no código, a ciência do software fornece uma variedade de métricas que podem ser usadas para avaliar a qualidade do programa.

Poucas métricas de produto têm sido propostas para uso direto em teste e manutenção de software. No entanto, muitas outras podem ser empregadas para orientar o processo de teste e como mecanismo para avaliar a sustentabilidade de um programa de computador. Uma grande variedade de métricas orientadas a objeto tem sido proposta para avaliar a testabilidade de um sistema orientado a objeto.

PROBLEMAS E PONTOS A PONDERAR

23.1. A teoria de medidas é um tópico avançado que tem grande influência sobre as métricas de software. Usando [Zus97], [Fen91], [Zus90] ou fontes baseadas na Web, escreva um pequeno artigo que destaque os principais princípios da teoria das medidas. Projeto individual: desenvolva uma apresentação sobre o assunto e apresente para a sua classe.

23.2. Por que não é possível desenvolver uma métrica única e totalmente abrangente para complexidade de programa ou qualidade de programa? Tente criar uma medida ou métrica utilizada na prática que contrarie os atributos de métricas eficazes de software definidas na Seção 23.1.5.

23.3. Um sistema tem 12 entradas externas, 24 saídas externas, responde a 30 diferentes consultas externas, gerencia 4 arquivos de lógica interna e estabelece interface com 6 diferentes sistemas legados (6 EIFs). Todos esses dados são de complexidade média e o sistema global é relativamente simples. Calcule FP para o sistema.

23.4. O software para o Sistema X tem 24 requisitos funcionais e 14 requisitos não funcionais. Qual a peculiaridade dos requisitos? A totalidade?

23.5. Um grande sistema de informações tem 1.140 módulos. Há 96 módulos que executam funções de controle e coordenação e 490 módulos cuja função depende de processamento

anterior. O sistema processa aproximadamente 220 objetos de dados tendo cada um deles em média três atributos. Há 140 itens distintos de base de dados e 90 diferentes segmentos de base de dados. Por fim, 600 módulos têm pontos distintos de entrada e saída. Calcule o DSQI para esse sistema.

23.6. Uma classe **X** tem 12 operações. A complexidade ciclomática foi computada para todas as operações no sistema orientado a objeto, e o valor médio da complexidade de módulo é 4. Para a classe **X**, a complexidade para as operações 1 a 12 é 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectivamente. Calcule os métodos ponderados por classe.

23.7. Desenvolva uma ferramenta de software que calculará a complexidade ciclomática para um módulo de linguagem de programação. Você pode escolher a linguagem.

23.8. Desenvolva uma pequena ferramenta de software que executará uma análise Halstead sobre código-fonte de linguagem de programação de sua escolha.

23.9. Um sistema legado tem 940 módulos. A última versão requeria que 90 desses módulos fossem alterados. Além disso, 40 novos módulos foram acrescentados e 12 módulos antigos foram removidos. Calcule o índice de maturidade do software para o sistema.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Há uma quantidade surpreendentemente grande de livros dedicados às métricas de software, embora a maioria deles focalize métricas de processo e projeto não incluindo métricas de produto. Lanza e seus colegas (*Object-Oriented Metrics in Practice*, Springer, 2006) discutem as métricas orientadas a objeto e seu uso para analisar a qualidade de um projeto. Genero (*Metrics for Software Conceptual Models*, Imperial College Press, 2005) e Ejiogu (*Software Metrics*, Book-Surge Publishing, 2005) apresentam uma grande variedade de métricas técnicas para casos de uso, modelos UML e outras representações de modelagem. Hutcheson (*Software Testing Fundamentals: Methods and Metrics*, Wiley, 2003) aborda um conjunto de métricas para teste. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2d ed., 2002), Fenton e Pfleeger (*Software Metrics: A Rigorous and Practical Approach*, Brooks-Cole Publishing, 1998) e Zuse [Zus97] escreveram tratamentos completos sobre métricas de produto.

Os livros de Card e Glass [Car90], Zuse [Zus90], Fenton [Fen91], Ejiogu [Eji91], Moeller e Paulish (*Software Metrics*, Chapman e Hall, 1993), e Hetzel [Het93] tratam de métricas de produto com algum detalhe. Oman e Pfleeger (*Applying Software Metrics*, IEEE Computer Society Press, 1997) editaram uma antologia de importantes artigos sobre métricas de software.

Métodos para estabelecer um programa de métricas e seus princípios subjacentes para medidas de software são considerados por Ebert e seus colegas (*Best Practices in Software Measurement*, Springer, 2004). Shepperd (*Foundations of Software Measurement*, Prentice-Hall, 1996) também trata da teoria de medidas com algum detalhe. As pesquisas atuais são apresentadas no *Proceedings of the Symposium on Software Metrics* (IEEE, publicado anualmente).

Um resumo abrangente de dezenas de métricas úteis de software é apresentado na [IEE93]. Em geral, foi resumida uma discussão de cada uma das métricas com as “primitivas” (medidas) essenciais necessárias para computar a métrica e as relações apropriadas para efetuar a computação. Um apêndice fornece discussão e muitas referências.

Whitmire [Whi97] apresenta um tratamento abrangente e matematicamente sofisticado das métricas orientadas a objeto. Lorenz e Kidd [Lor94] e Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1996) realizam tratamentos dedicados às métricas orientadas a objeto.

Há disponível na Internet uma grande variedade de fontes de informação sobre métricas de software. Uma lista atualizada das referências na Web, relevantes às métricas de software, pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

GERENCIAMENTO DE PROJETOS DE SOFTWARE

Nesta parte de *Engenharia de Software: Uma Abordagem Profissional* você aprenderá as técnicas de gerenciamento necessárias para planejar, organizar, monitorar e controlar projetos de software. Estas questões são tratadas nos capítulos que seguem:

- Como as pessoas, os processos e os problemas devem ser gerenciados durante um projeto de software?
- Como as métricas de software podem ser usadas para gerenciar um projeto de software e o processo de software?
- Como uma equipe de software pode gerar estimativas confiáveis de trabalho, custo e duração do projeto?
- Que técnicas podem ser usadas para avaliar os riscos que podem ter um impacto sobre o sucesso do projeto?
- Como um gerente de projeto de software seleciona um conjunto de tarefas de engenharia de software?
- Como é criado um cronograma de projeto?
- Por que a manutenção e a reengenharia são tão importantes para os gerentes de engenharia de software e os profissionais?

Uma vez respondidas essas questões, você estará mais bem preparado para gerenciar projetos de software de forma a obter um produto de alta qualidade entregue no prazo.

24

CONCEITOS DE GERENCIAMENTO DE PROJETO

CONCEITOS-CHAVE

aplicações críticas (práticas)	579
coordenação e comunicação	573
decomposição do problema	574
equipes ágeis	572
equipe de software	570
escopo de software	574
líderes de equipe..	569
o princípio W ³ HH. .	578
partes interessadas.....	569
pessoas	568
produtos.....	574
projeto.....	577

No prefácio de seu livro sobre gerenciamento de projeto de softwares, Meller Page-Jones faz uma afirmação que pode ser divulgada por muitos consultores da engenharia de software:

Visitei dúzias de lojas, boas e ruins; observei dezenas de gerenciadores de processamento de dados e, novamente, bons e ruins. Com muita frequência, horrorizado enquanto esses gerentes dedicavam esforços em projetos que eram verdadeiros pesadelos, contorciam-se em prazos impossíveis ou entregavam sistemas que ultrajavam seus usuários e continuavam a devorar um bocado de tempo de manutenção.

O que Page-Jones descreve são sintomas que resultam em um leque de problemas técnicos e de gerenciamento. Entretanto, se um exame pós-morte fosse feito para todo projeto, iria se chegar a um tema constante: o gerenciamento do projeto estava fraco.

Neste e nos capítulos 25 a 29, serão apresentados conceitos que conduzem a um gerenciamento de projetos efetivo. Aqui, trataremos os princípios e os conceitos básicos do gerenciamento de projetos. No capítulo 25, abordaremos a medição de projeto e de processo, base para a tomada de decisões de um gerenciamento efetivo. Técnicas utilizadas para a estimação de custos estão no Capítulo 26. O 27 auxiliará a definição de um cronograma realístico do projeto. As atividades de gerenciamento que levam a uma efetiva monitoração

PANORAMA

O que é? Embora muitos de nós (em momentos mais críticos) adotemos a visão de Dilbert, ainda resta uma atividade bastante útil quando sistemas e projetos computacionais são desenvolvidos. Gerenciamento de projeto envolve planejamento, monitoração e controle de pessoas, processos e eventos que ocorrem à medida que o software evolui desde os conceitos preliminares até sua disponibilização, operacional e completa.

Quem realiza? De certa forma, todas as pessoas gerenciam, mas o escopo das atividades de gerenciamento varia entre os envolvidos de um projeto de software para outro. Um engenheiro de software gerencia suas atividades diárias, planejando, monitorando e controlando as tarefas técnicas. Os gerenciadores de projetos planejam, monitoram e controlam o trabalho de uma equipe de engenheiros de software. Já um gerente sênior coordena a interface entre o “lado comercial” e os profissionais de software.

Por que é importante? Desenvolvimento de software computacional é uma tarefa complexa, principalmente se envolver muitas pessoas trabalhando por um tempo relativamente longo. Por isso os projetos de software precisam ser gerenciados.

Quais são as etapas envolvidas? Entenda os 4 Ps: Pessoas, Produto, Processo e Projeto. As

pessoas devem ser organizadas para o trabalho de desenvolvimento de forma efetiva. A comunicação com o cliente e com outros interessados deve ocorrer para que o escopo e os requisitos do produto sejam compreendidos. Deve ser selecionado um projeto adequado para as pessoas e para o produto. O projeto deve ser planejado com base na estimativa do esforço e do prazo para a realização das tarefas: definindo artefatos, estabelecendo pontos de verificação (checagem) de qualidade e identificando mecanismos para monitorar e controlar o trabalho no plano de projeto.

Qual é o artefato? Assim que as atividades de gerenciamento iniciam, faz-se um plano de projeto. Define-se o processo e as tarefas a ser conduzidas, as pessoas que realizarão o trabalho e os mecanismos de avaliação de riscos, do controle das alterações e de avaliação de qualidade.

Como garantir que o trabalho foi realizado corretamente? Nunca se está completamente seguro de que o plano de projeto está correto até que se entregue um produto de alta qualidade, no prazo e dentro do orçamento. Entretanto, um gerente de projeto age corretamente quando encoraja o pessoal de desenvolvimento a trabalhar em conjunto como uma verdadeira equipe, concentrando-se nas necessidades do cliente e na qualidade do produto.

e mitigação de riscos estão no Capítulo 28. Por fim, o 29 considera a manutenção, a reengenharia e discute elementos (itens) de gerenciamento encontrados quando se lida com sistemas legados.

24.1 O ESPECTRO DE GERENCIAMENTO

Gerenciamento efetivo de desenvolvimento de software tem um foco nos 4 Ps: Pessoas, produto, processo e projeto. Essa ordem não é arbitrária. O gerente que se esquecer de que o trabalho do engenheiro de software consiste em esforço humano nunca terá sucesso no gerenciamento de projeto. Da mesma forma, aquele que falha no encorajamento amplo para a comunicação entre os envolvidos, bem cedo, no início da elaboração de um produto, corre o risco de desenvolver uma solução elegante para o problema errado. Um gerente que preste pouca atenção ao processo, arrisca-se a inserir métodos e ferramentas técnicas competentes em um vácuo. Aquele que embarcar sem um plano de projeto sólido compromete o sucesso do projeto.

24.1.1 Pessoal

Desde os anos 1960, debate-se acerca da valorização da cultura de ter pessoal de desenvolvimento motivado e de alto nível. Realmente, recursos humanos é um fator de tal importância que o Software Engineering Institute (SEI) desenvolveu um modelo para a maturidade e capacidade dos recursos humanos – o People-CMM (*People Capability and Maturity Model* – Modelo de Capacidade e Maturidade para Recursos Humanos) – em reconhecimento ao fato de que: “Toda organização precisa aprimorar continuamente sua habilidade para atrair, desenvolver, motivar, organizar e reter a força de trabalho necessária para atingir os objetivos estratégicos de seus negócios”[Cur 01].

O People-CMM define as seguintes práticas-chave para o pessoal de software: formação de equipe, comunicação, ambiente de trabalho, gerenciamento do desempenho, treinamento, compensação, análise de competência e de desenvolvimento, desenvolvimento de carreira, do grupo de trabalho, da cultura de equipe e de outros mais.

Em organizações que conseguem altos níveis de maturidade e capacidade, o People-CMM tem maior probabilidade de implementar práticas de gerenciamento de software efetivos.

O People-CMM é um parceiro para o modelo de integração para maturidade e capacidade em software: o SW – CMMi (Capítulo 30) conduz as organizações para a criação de um processo de software maduro. Os itens relacionados ao gerenciamento de recursos humanos e à estruturação dos projetos de software serão discutidos posteriormente, neste capítulo.

24.1.2 O produto

Antes de traçarmos um plano de projeto, devemos estabelecer os objetivos do produto e seu escopo, considerar as soluções alternativas e identificar as restrições técnicas e de gerenciamento. Sem tais informações, é impossível definir de forma razoável (e precisa) a estimativa de custo, a avaliação efetiva dos riscos, a análise realística das tarefas do projeto ou um cronograma gerenciável do projeto que forneça a indicação significativa de progresso das atividades.

Como desenvolvedores, devemos nos reunir com os interessados no software para definir os objetivos e o escopo do produto. Em muitos casos, tal atividade se inicia como parte da engenharia do sistema ou de engenharia do processo de negócio e continua como a 1ª etapa da Engenharia de Requisitos do software (Capítulo 5).

Os objetivos identificam as metas gerais do produto (do ponto de vista dos interessados) sem considerar como tais metas serão alcançadas. O escopo identifica os principais dados, funções e comportamentos que caracterizam o produto e, mais importante, tenta mostrar as fronteiras e limitações dessas características de maneira quantitativa.

Uma vez entendidos os objetivos e o escopo, consideram-se soluções alternativas. Apesar de se discutir muito pouco os detalhes, as alternativas capacitam os gerentes e desenvolvedores

a selecionar a melhor abordagem, dadas as restrições impostas pelos prazos de entrega, restrições orçamentárias, disponibilidade de pessoal, interfaces técnicas e uma miríade de outros fatores.



Aqueles que aderem à filosofia do processo ágil (Capítulo 3) argumentam que ele é mais improdutivo que os outros. Isso pode ser verdade, mas ainda sim têm um processo, e um software de engenharia ágil ainda requer disciplina.

24.1.3 O processo

Um processo de software (Capítulos 2 e 3) fornece a metodologia por meio da qual um plano de projeto abrangente para o desenvolvimento de software pode ser estabelecido. Poucas atividades metodológicas são aplicáveis a todos os projetos de software, independentemente do seu tamanho e complexidade.

Uma quantidade de diferentes conjuntos de atividades-tarefas, pontos de controle, artefatos de software e pontos de garantia de qualidade possibilitam que as atividades metodológicas sejam adaptadas às características do projeto de software e aos requisitos de equipe. Por fim, as atividades de apoio, como Garantia de Qualidade de Software, Gerenciamento de Configuração e de Medições, sobrepõem-se ao modelo do processo. Estas são independentes de quaisquer das atividades metodológicas e ocorrem ao longo do processo.

24.1.4 O projeto

Empregam-se projetos com planejamento e com controle por uma única e principal razão: é a única maneira de administrar a complexidade. E, mesmo assim, as equipes de software têm de se esforçar. Em um estudo de 250 grandes projetos de software entre 1998 e 2004, Caper Jones [Jon 04] constatou que “cerca de 25 obtiveram sucesso em cumprir cronograma, custos e objetivos quanto à qualidade. Em torno de 50 apresentaram atrasos em, no mínimo, 35% retardamentos sérios, enquanto 175 projetos tiveram uma experiência de atrasos e retardamentos sérios, ou ainda não conseguiram terminá-lo”. Apesar de, atualmente, a taxa de sucesso nos projetos de software possa ter melhorado de algum modo, a taxa de falhas em projeto permanece mais alta do que deveria¹.

Para evitar falha de projeto, o gerente e engenheiros que desenvolvem o produto devem evitar uma série de sinais de alertas comuns, devem entender os fatores críticos de sucesso que conduzem ao bom gerenciamento e desenvolver uma abordagem de senso comum no que se referir a planejamento, monitoramento e controle de projeto. Cada um desses itens é discutido na Seção 24.5 e no capítulo seguinte.

24.2 As Pessoas

Em um estudo publicado pelo IEEE [Cur 88], os vice-presidentes de engenharia de três principais companhias de tecnologia foram questionados quanto ao fator mais importante que contribui para o sucesso de um projeto de software. Eles responderam da seguinte maneira:

Vp 1: Suponha-se que se tenha de selecionar um único item no ambiente que seja mais importante. Eu diria que não são as ferramentas usadas, são as pessoas.

Vp 2: O ingrediente mais importante neste projeto foi reunir pessoas espertas... Muito poucas coisas a mais importam na minha opinião... O mais importante para um projeto é selecionar a equipe... O sucesso da organização do desenvolvimento de software está bastante associado à habilidade de recrutar bom pessoal.

Vp 3: A única regra que tenho no gerenciamento consiste em assegurar que possa reunir bom pessoal – bem como cultivá-los – e propiciar ambiente no qual os bons profissionais possam produzir.

¹ Dadas essas estatísticas, é razoável questionar como os impactos computacionais evoluem exponencialmente. Parte da resposta, penso eu, é que um substancial número dos projetos que falham nas primeiras tentativas é preconcebido com falhas. Clientes perdem interesse muito rapidamente (porque o que eles pediram não é tão importante quanto acharam que era em princípio), e os projetos são cancelados.

Realmente, esses são testemunhos convincentes quanto à importância do pessoal no processo de engenharia de software. Ainda assim, dos vice-presidentes de engenharia ao desenvolvedor mais simples, frequentemente consideram pessoal como um privilégio. Os gerentes afirmam (como o fez o grupo anterior) que pessoal é prioritário, entretanto, suas ações rebaixam suas palavras. Na próxima seção examinam-se os interessados que participam do processo de software e a maneira pela qual são organizados para desempenhar ações efetivas de engenharia de software.

24.2.1 Os interessados (comprometidos)

O processo de software (e todo o projeto de software) é formado por interessados que podem ser categorizados em um dos cinco grupos:

1. *Gerentes seniores* – definem os itens de negócio e, com frequência, exercem influência significativa no projeto.
2. *Gerentes (técnicos) de projetos* devem planejar, motivar, organizar e controlar os programadores que executam o trabalho em software.
3. *Programadores* devem ter habilidades técnicas necessárias para desenvolver a engenharia de um produto ou aplicativo de software.
4. *Clientes* especificam os requisitos para o software a ser submetidos ao processo de engenharia e outros envolvidos que têm interesses periféricos no produto final.
5. *Usuários finais* interagem com o software uma vez liberado para uso operacional, em ambiente de produção.

Todo projeto de software é composto por pessoas que se enquadram nessa taxonomia². Para ser efetiva, a equipe do projeto deve estar organizada para maximizar cada capacidade e habilidade dos profissionais. E essa é a tarefa do líder da equipe.

24.2.2 Líderes de equipe

Gerenciamento de projeto é uma atividade intensiva de pessoal, e, por essa razão, programadores competentes em geral resultam em maus líderes de equipe. Apenas não possuem a mistura certa de habilidade com pessoas. Ainda assim, como Edgemon afirma: “infelizmente e, com demasiada frequência, parece que, os indivíduos só assumem o papel de gerente de projeto e se tornam gerentes de projetos por acidente”. [Ed 95].

Em um excelente livro sobre liderança técnica, Jerry Weinberg [Wei 86] sugere um modelo MOI de liderança:

Motivação: a habilidade para encorajar o pessoal técnico a produzir com o melhor da sua habilidade.

Organização: a habilidade para moldar os processos já existentes (ou inventar novos) que irão capacitar o conceito inicial para ser traduzido num produto final.

Ideias ou inovação: a habilidade de encorajar pessoas para criar e ser criativas mesmo quando estiverem trabalhando de acordo com padrões estabelecidos para um produto ou aplicativo de software específico.

Weinberg sugere que líderes de projeto bem-sucedidos aplicam um estilo de gerenciamento de solução de problemas. Isto é, um gerente de projeto de software deve se concentrar em entender o problema a ser resolvido, administrando o fluxo de ideias e, ao mesmo tempo, deixar claro para todos da equipe (por meio de palavras e, muito mais importante, de ações) que a qualidade conta muito e que não deve ser comprometida.

Uma outra visão [Edg 95] das características que definem um efetivo gerenciamento de projeto concentra-se em quatro aspectos-chave:

 O que buscamos ao selecionar alguém para liderar um projeto de software?

2 Quando se desenvolvem aplicações para web (*WebApps*), outras pessoas não especialistas em software são envolvidas na criação de conteúdos.

“Resumindo, um líder é aquele que sabe para onde quer ir, e se levanta e vai.”

John Erskine

Solução de problema. Um gerente de projeto eficaz sabe diagnosticar itens técnicos e organizacionais que são os mais relevantes, sistematicamente estrutura uma solução ou motiva apropriadamente outros desenvolvedores a buscar a solução, põe em prática as lições aprendidas de projetos anteriores para novas situações e permanece suficientemente flexível para mudar de direção, caso as tentativas iniciais para a solução de problemas tenham sido infrutíferas.

Identidade gerencial. Um bom gerente de projeto deve assumir a reponsabilidade do projeto. Deve ter a confiança para assumir o controle quando necessário e deve assegurar que permitirá ao pessoal técnico seguir os seus instintos.

Realizações. Um gerente competente deve recompensar iniciativas e realizações para otimizar a produtividade de uma equipe. Deve demonstrar por meio de seus próprios atos que decisões por riscos controlados não serão punidas.

Formação de equipe e de influência. Um gerente efetivo deve ser capaz de “ler” pessoas, deve ser capaz de compreender sinais verbais e não verbais e reagir às necessidades das pessoas que estão enviando esses sinais. O gerente deve permanecer sob controle em situações de alto estresse.

24.2.3 Equipe de software

Existem praticamente tantas estruturas organizacionais humanas para desenvolvimento de software quantas organizações que desenvolvem software. Para melhor ou pior, a estrutura organizacional não pode ser facilmente modificada. Preocupações com os efeitos práticos e políticos da mudança organizacional não fazem parte do escopo de responsabilidade do gerente de projeto de software. Entretanto, a organização do pessoal diretamente envolvido em um novo projeto está sob a perspectiva do gerente do projeto.

A melhor estrutura de equipe depende do estilo de gerenciamento das organizações, quantidade de pessoas na equipe, seus níveis de habilidade e do grau de dificuldade geral do problema. Mantei [man 81] descreve sete fatores que devem ser considerados ao planejarmos a estrutura da equipe de engenharia de software:

- Grau de dificuldade do problema a ser solucionado
- Extensão do(s) programa(s) resultante(s) em linhas de códigos ou pontos de função
- Tempo que uma equipe trabalhará em conjunto (tempo de vida da equipe)
- Grau de modularização do problema
- Qualidade e confiabilidade do sistema a ser construído
- Rigidez das datas de entregas
- Grau de sociabilidade (comunicação) requerido para o projeto

Constantine [Con 93] sugere quatro “paradigmas organizacionais” para equipes de engenharia de software:

1. O *paradigma fechado* estrutura a equipe em uma hierarquia de autoridade tradicional. Tais equipes podem trabalhar bem em produção de software bastante similar a esforços de épocas passadas, mas se mostrarão menos propícias a ser inovadoras trabalhando sob o paradigma fechado.
2. O *paradigma randômico* estrutura a equipe de forma livre e depende da iniciativa individual de seus membros. Quando for requerida inovação ou avanço tecnológico, as equipes que seguem esse paradigma irão se distinguir, mas poderão ter de se esforçar quando requisitada uma “performance ordenada”.
3. O *paradigma aberto* atém-se a estruturar a equipe de maneira que consiga alguns dos controles associados com o paradigma fechado, mas também muito da inovação que ocorre ao

“Nem todo grupo é uma equipe, nem toda equipe é efetiva”

Glenn Parker

Quais fatores devem ser considerados quando a estrutura de uma equipe de software já está estabelecida?

Quais são as opções quando se tem de definir a estrutura de uma equipe de software?

usar o paradigma randômico. O trabalho é feito de forma colaborativa, com forte comunicação e tomada de decisão baseada no consenso, executando com as características marcantes das equipes de paradigma aberto. As estruturas das equipes de paradigmas abertos são bem adequadas para a solução de problemas complexos, mas não conseguem desempenhar tão bem quanto outras equipes.

4. O *paradigma sincronizado* baseia-se na compartimentalização natural de um problema e organiza os membros da equipe a trabalhar nas partes do problema com pouca comunicação entre si.

“Se deseja ser incrementalmente melhor, seja competitivo. Se quer ser exponencialmente melhor, seja cooperativo.”

Autor desconhecido

Como uma nota de rodapé histórica, uma das mais antigas organizações de equipe de software foi paradigma de uma estrutura fechada denominada originalmente de equipe com um programador-chefe (principal). Essa estrutura foi primeiramente proposta por Harlan Mills e descrita por Baker [Bak 72]. O núcleo da equipe era composto de um engenheiro sênior (o programador-chefe principal), que planeja, coordena e faz a revisão de todas as atividades técnicas da equipe, o pessoal técnico (normalmente de duas a cinco pessoas), que conduz as atividades de análise e de desenvolvimento, e um engenheiro reserva que dá suporte ao engenheiro sênior em suas atividades e pode substituí-lo com perdas mínimas de continuidade. O programador-chefe (principal) pode ter a seus serviços um ou mais especialistas (por exemplo, perito em telecomunicações, desenvolvedor de banco de dados), equipe de suporte (por exemplo, codificadores técnicos, pessoal de escritório) e um bibliotecário de software.

Como um contraponto para a estrutura da equipe de programadores principais (líderes), o paradigma randômico de Constantine [Con 93] sugere a primeira equipe criativa, cuja abordagem de trabalho pode ser mais bem denominada de “anarquia inovativa”. Embora a abordagem de espírito livre para o trabalho de software tenha apelo, a energia da criatividade direcionada para uma equipe de alta performance deve ter o objetivo central de uma organização de engenharia de software. Para obter uma equipe de alta performance:

- Os membros da equipe devem confiar uns nos outros.
- A distribuição de habilidades deve ser adequada ao problema.
- Estrelismos devem ser excluídos da equipe para manter a coesão do grupo.

 O que é equipe “consistente”?

Seja qual for a organização da equipe, o objetivo em todo o gerenciamento do projeto consiste em ajudar a montar uma equipe que apresente coesão. Em seu livro *Peopleware*, De Marco e Listes [DeM 98] discorrem sobre esse tema:

Há uma tendência em se utilizar a palavra equipe de forma constante e vaga na área de negócios, denominando qualquer grupo de profissionais designados a trabalhar juntos de “equipe”. Entretanto, muitos deles não se assemelham a equipes. Não há uma definição comum de sucesso nem um espírito de equipe identificável. O que falta é um fenômeno que se denomina *consistência*.

Uma equipe consistente é um grupo de pessoas tão fortemente unidas que o todo é maior do que a soma das partes. Uma vez que uma equipe comece a ser consistente, a probabilidade de sucesso segue em caminho ascendente. A equipe pode disparar para um caminho de sucesso...

Não é preciso gerenciá-la do modo tradicional e, com certeza, não precisará ser motivada. Ela adquire velocidade e ímpeto.

 Por que motivo equipes falham em ser consistentes?

De Marco e Lister sustentam que os membros de equipes consistentes são mais significativamente produtivos e mais motivados do que a média. Compartilham de um objetivo comum, de uma cultura comum e, em muitos casos de um senso de elitização que os torna únicos.

Mas nem todas as equipes tornam-se consistentes. De fato, muitas sofrem do que Jackman [Jac 98] denomina de “toxicidade de equipe”. Ela define cinco fatores que fomentam um ambiente potencialmente tóxico da equipe: uma atmosfera de trabalho frenética; alto grau de frustração que causa atrito entre os membros da equipe; um processo de software fragmentado ou pobremente coordenado; uma definição nebulosa dos papéis dentro da equipe de software; e contínua e repetida exposição a falhas.

“Ou faça ou não faça. Não existe o tentar”.

Yoda,
personagem de
Star Wars

Para evitar um ambiente de trabalho frenético, o coordenador de projeto deve estar certo de que a equipe tem acesso a todas as informações necessárias para realizar o trabalho e de que as metas e objetivos prioritários (papéis), uma vez estabelecidos, não devem ser alterados a menos que absolutamente necessário. Uma equipe pode evitar frustrações se lhe for oferecida, tanto quanto possível, responsabilidade para tomada de decisão. Um processo inapropriado (por exemplo, tarefas pesadas ou desnecessárias ou artefatos mal selecionados) pode ser evitado por meio da compreensão do produto a ser desenvolvido, das pessoas que realizam o trabalho e pela permissão para que a equipe selecione o modelo do processo. A própria equipe deve estabelecer seus próprios mecanismos de responsabilidades (revisões técnicas³ são excelentes meios para conseguir isso) e deve definir uma série de abordagens para correções quando um membro falhar em suas atribuições. E, finalmente, a chave para evitar uma atmosfera de derrota consiste em estabelecer técnicas baseadas nas equipes voltadas para realimentação (*feedback*) e resolução de problemas.

Somando-se às cinco toxinas descritas por Jackman, uma equipe de software frequentemente dispense esforços com as diferentes características de seus membros. Uns são extrovertidos, outros introvertidos. Uns coletam informações intuitivamente, destilando conceitos amplos de fatos disparatados. Outros processam informações linearmente, coletando e organizando detalhes mínimos dos dados fornecidos. Alguns se sentem confortáveis tomando decisões apenas quando um argumento lógico e ordenado for apresentado. Outros são intuitivos, acostumados a tomar decisões baseadas em percepções. Certos desenvolvedores querem um cronograma detalhado, preenchido por tarefas organizadas que os tornem aptos para atingir proximidade com elementos de projeto. Outros ainda preferem um ambiente mais espontâneo no qual resultados e questões abertas já estarão bons.

Alguns trabalham arduamente para conseguir que as etapas sejam concluídas bem antes da data estabelecida, evitando, portanto, estresse na medida em que se aproxima a data-limite, enquanto outros são energizados pela correria em fazer até a data e minutos-limite.

Uma discussão detalhada sobre a psicologia envolvida nessas características e formas pelas quais um hábil líder de equipe pode auxiliar pessoas com traços opostos a trabalhar juntas está além do escopo deste livro⁴. Entretanto, é importante notar que o reconhecimento das forças humanas é o primeiro passo em direção à criação de equipes consistentes.

24.2.4 Equipes ágeis

Ao longo da última década, o desenvolvimento de software ágil (Capítulo 13) tem sido indicado como o antídoto para muitos problemas que se alastraram nas atividades de projeto de software. Relembrando, a filosofia ágil enfatiza a satisfação do cliente e a entrega prévia incremental de software, pequenas equipes de projetos altamente motivadas, métodos informais, mínimos artefatos de engenharia de software e total simplicidade de desenvolvimento.

A pequena, altamente motivada equipe de projeto, também denominada de *equipe ágil*, adota muitas das características das equipes de software bem-sucedidas, discutidas na seção anterior e evita muito das toxinas geradoras de problemas. Entretanto, a filosofia ágil enfatiza competência individual (membro da equipe) casada com colaboração em grupo como fatores críticos de sucesso para a equipe. Cockburn e HighSmith [Coc01a] observam isso ao escreverem:

Se as pessoas do projeto forem boas o suficiente, podem usar praticamente qualquer processo e realizar a sua missão. Se elas não forem boas o suficiente, nenhum processo irá reparar a sua inadequação. “Pessoas são o trunfo do processo” é uma forma de dizer isso. Entretanto, falta de suporte ao desenvolvedor e ao usuário pode matar um projeto – “política é o trunfo de pessoas”. Suporte inadequado pode fazer com que mesmo os bons fracassem na realização de seus trabalhos.

Para uso das competências de cada membro da equipe, para fomentar colaboração efetiva ao longo do projeto, equipes ágeis se auto-organizam. A equipe que se auto-organiza não mantém,

PONTO-CHAVE

Uma equipe ágil é aquela que se organiza, que tem autonomia para planejar e tomar decisões técnicas.

3 Revisões técnicas são tratadas em detalhes no Capítulo 15.

4 Uma excelente introdução a essas questões no que se refere a equipes de projeto de software pode ser encontrada em [Fer 98].

necessariamente, uma estrutura de equipe única, mas usa elementos da aleatoriedade de Constantine, paradigmas abertos e de sincronicidade discutidos na Seção 24.2.3.

Muitos modelos ágeis de processo (por exemplo, Scrum) dão à equipe ágil autonomia para fazer o gerenciamento do projeto e para decisões técnicas necessárias à conclusão do trabalho. O planejamento é mantido em um nível mínimo, e a equipe tem a permissão para selecionar sua própria abordagem (por exemplo, processo, método, ferramentas), limitada somente pelos requisitos de negócio e pelos padrões organizacionais. Conforme o projeto prossegue, a equipe se auto-organiza, concentrando-se em competências individuais para maior benefício do projeto em determinado ponto de cronograma. Para tanto, uma equipe ágil pode realizar reuniões de pessoal diariamente para coordenar e sincronizar as atividades que devem ser realizadas para aquele dia.

Com base na informação obtida durante essas reuniões, a equipe adapta sua abordagem para incrementar o trabalho. A cada dia, contínuas auto-organizações e colaboração conduzem a equipe em direção a um incremento de software completo.

24.2.5 Itens de comunicação e coordenação

“Propriedades coletivas nada mais são do que um imediatismo da ideia de que os produtos deveriam ser atribuídos à equipe (ágil), não a indivíduos que compõem a equipe.”

Jim Highsmith

Há muitas razões para que o projeto de software tenha problemas. A escala de muitos esforços em desenvolvimento é ampla, conduzindo a complexidade, confusão e dificuldades significativas na coordenação dos membros da equipe. Incertezas são comuns, resultando em contínua cadeia de alterações que racham a equipe de projeto. Interoperabilidade torna-se um aspecto-chave para muitos sistemas. Novo software deve se comunicar com os softwares existentes e se ajustar a restrições predefinidas impostas pelo sistema ou pelo produto.

Tais características do software moderno – escala, incerteza e interoperabilidade – são fatos da vida. Para lidar efetivamente com eles, devem-se estabelecer métodos efetivos para coordenar pessoas que realizam o trabalho. Para tanto, devem-se estabelecer mecanismos para comunicação formal e informal entre os membros de equipes. A comunicação formal é realizada por meio de “comunicação escrita, reuniões estruturadas e de outros canais de comunicação relativamente não interativos e impessoais” [Kra 95]. A comunicação informal é mais pessoal.

CASA SEGURA



Estrutura de time

Cena: Escritório de Doug Miller antes do início do projeto de software *CasaSegura*.

Atores: Doug Miller (coordenador/gerente da equipe de engenharia de software do *CasaSegura*), Vinod Ramann, Jamie Lazar e outros membros da equipe de engenharia de software do produto.

Conversa:

Doug: Vocês deram uma olhada no informativo preliminar do *CasaSegura* que o departamento de marketing preparou?

Vinod (balançando afirmativamente e olhando para seus companheiros de equipe): Sim, mas temos muitas dúvidas.

Doug: Vamos deixar isso de lado por um momento. Gostaria de conversar sobre como vamos estruturar a equipe, quem será responsável pelo quê?

Jamie: Estou realmente de acordo com a filosofia ágil, Doug. Acho que devemos ser uma equipe que se auto-organiza.

Vinod: Concordo. Devido ao cronograma apertado e o grau de incerteza e do fato de todos sermos realmente competentes (risos...), parece ser o caminho certo a tomar.

Doug: Tudo bem para mim, mas vocês conhecem o procedimento.

Jamie (solicitando e falando ao mesmo tempo): Tomamos decisões táticas acerca de quem faz o que e quando, mas é de nossa responsabilidade ter o produto pronto sem atraso.

Vinod: E com qualidade.

Doug: Exatamente. Mas lembrem-se de que há restrições. O marketing define os incrementos de software a ser desenvolvidos – consultando-nos, é claro.

Jamie: E?

Doug: E usaremos o UML como abordagem de modelagem.

Vinod: Mas mantenham documentações adicionais no mínimo (absoluto).

Doug: Quem manterá a ligação comigo?

Jamie: Decidimos que Vinod será o coordenador técnico – ele tem mais experiência, portanto, será o intermediário, mas sinta-se livre para conversar com qualquer um de nós.

Doug (rindo): Não se preocupe, farei isso.

Os membros de uma equipe de software compartilham ideias numa base *ad hoc*, solicitam ajuda conforme surgem os problemas e interagem uns com os outros diariamente.

24.3 O PRODUTO

Um gerente de projeto de software confronta-se com um dilema sempre que inicia um projeto. É preciso estimativas quantitativas e um plano organizado, entretanto informações sólidas não estão disponíveis ainda. Uma detalhada análise dos requisitos de software fornece as informações necessárias para as estimativas, mas as análises frequentemente levam semanas ou mesmo meses para estarem completas. Pior ainda, os requisitos podem ser fluidos, mudando regularmente conforme o projeto prossegue. Ainda assim, um planejamento é necessário “agora”! Gostando-se ou não, deve-se examinar o produto e o problema que se pretende solucionar logo no início do projeto. No mínimo, o escopo do produto deve ser estabelecido e delimitado.

24.3.1 Escopo de software

A primeira afinidade do gerenciamento do projeto de software consiste em determinar o escopo de software. Este é definido respondendo-se às seguintes questões:



Se puder estabelecer uma característica do software que pretende construir, liste-a como um risco de projeto (Capítulo 25).

Contexto. Como o software a ser desenvolvido se ajusta a um sistema maior, a um produto ou contexto de negócio e quais são as restrições impostas resultantes do contexto.

Objetivo da informação. Quais objetos de dados visíveis ao cliente são produzidos como saída de software? Quais objetos de dados são necessários como entrada?

Função e performance. Qual a função que o software desempenha para transformar os dados de entrada em dados de saída? Há quaisquer características especiais de desempenho a ser acessada?

O escopo do projeto de software não deve haver ambiguidades e deve ser compreensível tanto no nível gerencial quanto no nível técnico. Deve-se estabelecer o escopo de software. Isto é, dados quantitativos (por exemplo, número de usuários simultâneos, ambiente-alvo, tempo máximo de resposta) são estabelecidos explicitamente, restrições e/ou limitações (por exemplo, custo do produto restringe tamanho de memória) são determinadas e fatores mitigadores (por exemplo, algoritmos desejados são bem compreendidos e avaliados em Java) são descritos.

24.3.2 Decomposição do problema

A decomposição do problema, também chamada de *elaboração do problema* ou *particionamento*, consiste em atividade que ocupa o centro da análise de requisitos de software (Capítulos 6 e 7). Durante a atividade de escopo, não se busca decompor completamente o problema. De preferência, aplica-se a decomposição em duas áreas vitais: (1) na funcionalidade e no conteúdo (informação) que deve ser entregue; e (2) no processo que será utilizado para entregar o software.

As pessoas tendem a aplicar a estratégia de dividir para conquistar quando confrontadas por um problema complexo. Ao simplificarmos um problema complexo, este é particionado em pequenas questões mais gerenciáveis. Essa é a estratégia a aplicar no início do planejamento do projeto.

Funções de software, descritas no estabelecimento do escopo, são avaliadas e refinadas para proporcionar mais prioridades de detalhes logo no início das estimativas (Capítulo 26). Por serem estimativas, custos ou cronogramas são orientados pela funcionalidade, algum grau de decomposição é em geral útil. Do mesmo modo, conteúdo principal ou objetos de dados são decompostos em suas partes constituintes, propiciando compreensão razoável da informação a ser gerada pelo software.

Como exemplo, considere um projeto que irá desenvolver um produto de processador de texto. Entre os fatores importantes estão recursos de voz, bem como entrada por teclado virtual



Para elaborar um razoável planejamento de projeto, deve-se decompor o problema. Para tanto, utiliza-se uma lista de funções ou empregam-se, como base, casos de uso.

através de uma tela de toques múltiplos, recursos de edição/correção com cópias automáticas extremamente sofisticadas, capacidade de formatação de layout de página, indexação automática e tabela de conteúdos, entre outros. O gerente de projeto deve primeiro estabelecer um escopo de declarações e procedimentos que abrange esses recursos (e outras funções mais comuns como correções, alterações, gerenciamento de arquivos e produção de documentos). Por exemplo, o recurso de entrada por voz irá requerer que o produto tenha o recurso de “aprendizado” e “treinamento” pelo usuário? Especificamente, quais capacidades serão fornecidas para as funções de edição das cópias? Qual o grau de sofisticação da função de formatação de página, considerando o uso da tela multitoque e suas capacidades implícitas?

À medida que os parâmetros do escopo evoluem, ocorre um particionamento natural em primeiro nível. A equipe é notificada de que o departamento de marketing, ao conversar com clientes potenciais, detectou que as seguintes funções deveriam fazer parte do recurso de edição automática das cópias: (1) checagem de separação silábica; (2) checagem de gramática; (3) checagem de referência para documentos externos (por exemplo, uma referência a uma entrada bibliográfica é encontrada na lista de entrada da bibliografia?); (4) implementação de recursos de folhas de estilos que imponha consistência ao longo do documento; e (5) validação da referência quanto à seção e capítulo para documentos externos.

Cada um desses recursos representa uma subfunção a ser implementada no software. Podem ser refinados caso a decomposição (subdivisão e o particionamento) facilite o planejamento.

24.4 O PROCESSO

As atividades de modelagem (Capítulo 2) que caracterizam o processo de software são aplicáveis a todos os projetos de software. A dificuldade está em selecionar o modelo de processo apropriado ao software a ser desenvolvido (pelo processo de engenharia) pela sua equipe.

A equipe deve decidir qual modelo de processo será mais apropriado: (1) aos clientes que solicitaram o produto e aos profissionais que realizarão o trabalho; (2) às próprias características de produto; e ao ambiente de projeto no qual a equipe trabalhará.

Quando um modelo de processo é selecionado, a equipe define o planejamento preliminar do projeto com base no conjunto de atividades estabelecidas no modelo do processo. Uma vez definido o planejamento preliminar, inicia-se o particionamento (decomposição) do projeto. Ou seja, um planejamento completo, que reflita as tarefas de trabalho necessárias para preencher as atividades de modelagem, deve ser criado. Essas atividades serão abordadas nas seções seguintes e uma visão mais detalhada, apresentada no Capítulo 26.

24.4.1 Combinando o produto e o processo

O projeto começa por meio da combinação do produto com o processo. Cada função a ser desenvolvida por engenharia deve passar pela estrutura de atividades definidas pela organização responsável pelo software.

Suponha que a organização tenha adotado a estrutura de atividades genéricas **comunicação, planejamento, modelagem, construção e empacotamento dos programas executáveis** tratados no Capítulo 2. Os membros da equipe que trabalha em uma funcionalidade do produto aplicarão cada uma das atividades estruturadas para a função. Em essência, uma matriz similar à Figura 24.1 será criada. Cada função principal do produto (a figura mostra a função do software do processador de texto discutido anteriormente) será listada na coluna da esquerda. As atividades estruturadas serão relacionadas (indicadas) na parte superior das colunas. As tarefas de trabalho de engenharia de software (para cada atividade estruturada) serão incluídas nas linhas seguintes⁵. O trabalho do gerente de projeto e de outros membros da equipe será estimar as necessidades de recursos para cada célula da matriz, datas de início e de fim para as tarefas

5 As tarefas devem ser adaptadas às necessidades específicas do projeto, baseando-se em critérios de adaptação.

Agora, considere um projeto mais complexo, com um escopo mais amplo e um impacto comercial mais significativo. Tal projeto pode vir a requerer as seguintes tarefas para **comunicação**:

1. Revisão da solicitação do cliente.
2. Planejamento e agendamento de reuniões viabilizadas e formais com todos os envolvidos.
3. Realização de uma pesquisa para especificar a solução proposta e as abordagens existentes.
4. Preparação de um documento de trabalho e de uma agenda para a reunião formal.
5. Realização de reunião.
6. Desenvolvimento conjuntamente de miniespecificações que reflitam os dados, a funcionalidade e os fatores comportamentais do software. De forma alternativa, desenvolvimento de casos de uso que descrevam o software sob o ponto de vista do usuário.
7. Revisão de cada miniespecificações ou caso de uso para realizar correções, consistências e eliminação de ambiguidades.
8. Reunião de miniespecificações em um documento de escopo (bases).
9. Revisão do documento de escopo (bases) ou a coletânea de casos de uso com todos os envolvidos.
10. Alteração do documento de escopo ou de casos de uso conforme o necessário.

Ambos os projetos constituem as atividades **comunicação**, entretanto, a primeira equipe executa metade das tarefas de trabalho de engenharia.

24.5 O PROJETO

Para gerenciar um projeto de software com sucesso, deve-se compreender o que pode sair errado, de modo que ações planejadas evitem tais problemas. Em excelente artigo sobre projetos de software, John Reel [Ree 99] indica 10 sinais indicadores de que um projeto de sistemas de informações está em perigo:

 **Quais são os sinais de que um projeto de software está em perigo?**

1. O pessoal de software não compreende as necessidades de seus clientes.
2. O escopo do produto está parcialmente definido.
3. As alterações são mal gerenciadas/administradas.
4. A tecnologia escolhida muda.
5. As necessidades de negócio mudam (ou são mal definidas).
6. Os prazos estão fora da realidade.
7. Os usuários mostram-se resistentes.
8. O patrocínio é perdido (ou nunca foi propriamente obtido).
9. Faltam profissionais à equipe ou esta não possui pessoal com habilidades adequadas.
10. Gerentes e desenvolvedores evitam práticas e lições aprimoradas e aprendidas.

Não há tempo para parar e reabastecer, já estamos atrasados.

M. Cleron

Com frequência profissionais da indústria já estressados referem-se à regra 90-90 ao debaterem sobre projetos particularmente difíceis. Os primeiros 90% de um sistema absorvem 90% dos esforços e tempos alocados. Os 10% restantes consomem outros 90% de esforço e tempo alocados [Zah 94]. As situações que conduzem à regra 90-90 foram incluídas nos indicadores da lista anterior.

Porém, chega de negatividade! Como um gerente age para evitar os problemas mencionados? Reel [Ree 99] sugere uma abordagem de cinco partes para os projetos de software:

1. *Comece com o pé-direito.* Trabalhando arduamente (muito arduamente), é possível compreender o problema a ser solucionado e, então, estabelecer expectativas e objetivos realísticos

para todos os envolvidos no projeto. Isso é reforçado por meio da formação da equipe correta (Seção 24.2.3) e concedendo-lhe autonomia, autoridade e tecnologia necessárias para realizar o trabalho.

2. *Mantenha a velocidade (ímpeto).* Muitos projetos começam bem e depois desintegram lentamente. Para manter velocidade (ímpeto), o coordenador deve fornecer incentivos para que a rotatividade de pessoal fixe-se em um nível absolutamente mínimo. A equipe deve dar ênfase à qualidade em todas as tarefas que realiza e o coordenador sênior deve fazer todo o possível para posicionar-se fora do caminho da equipe⁷.
3. *Rastreie o andamento.* Em um projeto de software, o andamento é rastreado e mapeado como os artefatos (códigos-fonte, conjunto dos pacotes de testes) são produzidos e aprovados (usando-se as revisões técnicas) como parte de uma atividade de garantia de qualidade. Como acréscimo, o processo de software e as medições do projeto (Capítulo 25) podem ser coletados e utilizados para avaliar o progresso (andamento) em relação às médias desenvolvidas para a organização de desenvolvimento de software.
4. *Tome decisões com astúcia (rapidez).* Em essência, a decisão do gerente de projeto e da equipe de software deve ser a de “manter a simplicidade”. Sempre que possível, decida-se por utilizar software comercial ou por componentes e modelos de software existentes. Evite interfaces customizadas quando houver disponibilidade de abordagens-padrão. Fique atento aos riscos óbvios para evitá-los e decida-se por alocar maior prazo do que o necessário para tarefas arriscadas ou complexas (serão necessários todos os minutos). Somente as práticas vitais associadas à “integridade do projeto” são registradas aqui.
5. *Faça uma análise post-mortem.* Estabeleça um mecanismo consistente para extrair aprendizados de cada projeto. Avalie os cronogramas planejados e os realizados, as métricas de projetos de software coletadas e analisadas, obtenha feedback dos membros da equipe e dos clientes e registre por escrito.

“Um projeto é como uma rodovia. Alguns são simples e rotineiros, como dirigir até uma loja em dia ensolarado. Entretanto, a maioria dos que valem ser realizados parece mais com dirigir um caminhão à noite em uma serra cheia de curvas.”

Cem Kaner,
James Bach e
Bret Pettichord

24.6 O PRINCÍPIO W⁵HH

Em excelente artigo sobre projeto e processo de software, Barry Boehm [Boe 96] afirma: “É necessário um princípio organizacional que facilite a obtenção de planejamentos simples para projetos simples”. Boehm propõe uma abordagem voltada para os objetivos do projeto, marcos (pontos de referência) e cronogramas (agendas), responsabilidades, gerenciamento, abordagens técnicas e recursos necessários. Ele a denominou de *Princípio W⁵HH*, após uma série de perguntas que conduzem a uma definição das características-chave do projeto e do planejamento do projeto resultante:

 **Como definir as características-chave do projeto?**

Por que o sistema está sendo desenvolvido? Todos os interessados devem avaliar a validade das razões comerciais para o trabalho de software. O propósito justifica os gastos referentes a pessoal, tempo e dinheiro?

O que será feito? Define-se o conjunto de tarefas necessárias para o projeto.

Quando será feito? A equipe definirá o cronograma de projeto, identificando quando serão realizadas as tarefas e quando os pontos de referências (marcos) serão atingidos.

Quem será o responsável por uma função? Os papéis e responsabilidades de cada membro serão definidos.

Onde se posicionam organizacionalmente? Nem todas as situações e responsabilidades estão a cargo dos desenvolvedores de software. O cliente, os usuários e outros envolvidos também têm as suas responsabilidades.

⁷ A implicação dessa orientação é que se reduz a burocracia a um mínimo, reuniões extras são eliminadas e radicalismos ao processo e às regras do projeto são atenuados. A equipe deve ser auto-organizada e autônoma.

Como será realizado o trabalho técnica e gerencialmente? Uma vez estabelecido o escopo, deve-se definir uma estratégia técnica e gerencial.

Quanto cada recurso será necessário? A resposta a essa pergunta irá derivar-se de estimativas desenvolvidas (Capítulo 26), baseando-se nas respostas a questões anteriores.

O princípio W⁵HH de Boehm é aplicável não importando o tamanho ou complexidade do projeto. As questões apontadas fornecem excelente esquema de planejamento.

24.7 PRÁTICAS VITAIS

O Conselho de Airlie⁸ desenvolveu uma lista de “práticas de software vitais para gerenciamento baseado no desempenho”. Esses procedimentos são usados de forma consistente e considerados críticos para projetos de software altamente bem-sucedidos e por organizações cujo desempenho mínimo é consistentemente melhor que as médias da indústria [Air 99].

Práticas vitais⁹ incluem: gerenciamento de projeto baseado em métricas (Capítulo 25), custos empíricos e estimativas de cronogramas (Capítulos 26 e 27), acompanhamento de valorização (Capítulo 27), acompanhamento de defeitos em contrapartida com os objetivos de qualidade (Capítulos 14 a 16) e gerenciamento consciente de pessoal (Seção 24.2). Cada uma dessas práticas é mencionada ao longo das Partes 3 e 4 deste livro.

FERRAMENTAS DO SOFTWARE



Ferramentas de software para gerentes de projeto

As ferramentas listadas aqui são genéricas e se aplicam a uma larga escala de atividades realizadas por gerentes de projetos. Ferramentas específicas para o gerenciamento de projeto (por exemplo, ferramentas para elaboração de cronogramas, para estimativas e para análise de riscos) serão consideradas em capítulos posteriores.

Ferramentas representativas¹⁰:

O Software Program Manager's Network (www.spmn.com) desenvolveu uma ferramenta simples denominada *Project Control Panel*

(Painel de Controle de Projeto), que oferece aos gerentes de projeto uma indicação direta do status do projeto. A ferramenta tem indicadores (medidas/padrões) semelhantes a um painel (quadro) e é implementada com o Excel da Microsoft. Está disponível para download no site www.spmn.com/products_software.html.

Ganthead.com (www.ganthead.com/) desenvolveu um conjunto de listas úteis de verificações para gerentes de projetos.

Ittoolkit.com (www.ittoolkit.com) fornece uma coletânea de guias de planejamento de processo e modelos de processo e folhas de trabalho inteligentes, disponíveis no CD-ROM.

24.8 RESUMO

O gerenciamento de projeto de software é uma atividade de apoio da engenharia de software. Inicia-se antes de qualquer atividade técnica e prossegue ao longo da modelagem, construção e utilização do software.

Os quatro Ps (4Ps) têm influência substancial no gerenciamento do projeto de software – pessoas, produto, processo e projeto. As pessoas devem ser organizadas em equipes efetivas, motivadas para realizar um trabalho de software de alta qualidade e coordenada para uma comunicação efetiva. Requisitos de produto devem ser comunicados do cliente ao desenvolvedor, decompostos em partes constituintes e posicionados para a execução pela equipe de software. O processo deve ser adaptado às pessoas e aos produtos. Uma estrutura comum de processo é

8 O conselho Airlie foi formado por uma equipe de especialistas em engenharia de software e registrado pelo Departamento de Defesa dos Estados Unidos para ajudar a desenvolver um guia de melhores práticas para o gerenciamento de projeto de software e para engenharia de software. Mais informações encontram-se em www.swqual.com/newsletter/vol1/no3/vol1no3.html.

9 Práticas vitais adotadas aqui referem-se à integridade do projeto.

10 As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

selecionada, é aplicado um paradigma de engenharia de software apropriado e um conjunto de tarefas é escolhido para que o trabalho se realize. Por fim, deve-se organizar de uma forma que capacite a equipe de software a um trabalho bem-sucedido.

O elemento-chave de todos os projetos de software são os profissionais. Engenheiros de software podem ser organizados em uma série de diferentes estruturas de equipes que abarcam desde hierarquias de controle tradicional a equipes de “paradigma aberto”. Uma variedade de técnicas de comunicação e coordenação pode ser aplicada para dar suporte ao trabalho da equipe. Em geral, revisões técnicas e comunicação informal de pessoa a pessoa têm o maior valor para os desenvolvedores.

As atividades de gerenciamento de projeto englobam medições e métricas, estimativas e agendamento, análise de riscos, acompanhamento e controle. Cada um desses tópicos será considerado nos capítulos seguintes.

PROBLEMAS E PONTOS A PONDERAR

Baseando-se nas informações deste capítulo e na sua experiência, desenvolva os “dez mandamentos” para fortalecer o engenheiro de software. Ou seja, faça uma lista de dez princípios que guiarão os desenvolvedores a trabalhar com o máximo de potencial.

O People-CMM do Instituto de Engenharia de Software (SEI) adota uma visão organizada para as “áreas de processo-chave” (KPA) cultivando o bom pessoal de software. O seu instrutor irá lhe indicar uma KPA para análise e resumo.

Descreva três situações cotidianas nas quais o cliente e o usuário final são os mesmos. Descreva três situações nas quais eles são distintos.

As decisões tomadas por gerenciamento sênior podem ter impacto significativo na eficiência da equipe de engenharia de software. Forneça cinco exemplos ilustrativos em que isso seja verdadeiro.

Faça uma análise do livro do Weinberg [Wei 86] e um resumo de duas ou três páginas sobre os itens a considerar ao aplicarmos o modelo MOI.

Você foi nomeado gerente de projeto em uma organização de sistemas de informações. Sua tarefa é construir uma aplicação bastante similar a outras que sua equipe desenvolveu, embora esta seja maior e mais complexa. Requisitos foram completamente documentados pelo cliente. Qual estrutura de equipe você escolheria e por quê? Qual modelo de processo de software escolheria e por quê?

Você foi nomeado gerente de projeto em uma companhia de software. Sua tarefa é desenvolver algo inovador que combine hardware de realidade virtual com software “estado da arte”. Pelo fato de a competitividade pelo mercado de entretenimento doméstico ser intensa, há pressão significativa quanto à conclusão do trabalho. Qual estrutura de equipe escolheria e por quê? Qual modelo de processo de software escolheria e por quê?

Você foi nomeado para gerente de projeto em uma companhia de software. Seu trabalho é gerenciar o desenvolvimento da versão da próxima geração do seu amplamente utilizado processador de texto. Por haver competição intensa, prazos de entrega curtos foram estabelecidos e anunciados. Qual estrutura de equipe você escolhe e por quê? Qual modelo de processo de software você escolhe e por quê?

Você foi nomeado gerente de projeto de software em uma companhia que presta serviços para o setor de engenharia genética. Seu trabalho é administrar o desenvolvimento de um novo software que acelerará o ritmo de classificação de tipos de genes. O trabalho é orientado por Pesquisa e Desenvolvimento (P&D), mas o objetivo é produzir um produto para o próximo ano. Qual estrutura de equipe você escolhe? Qual estrutura de processo você escolhe? E por quê?

Foi-lhe solicitado desenvolver uma pequena aplicação que analise cada curso oferecido por uma universidade e emita relatórios sobre a média obtida no curso (para determinada turma). Faça uma declaração de escopo que englobe esse problema.

Faça uma decomposição funcional de nível 1 da função de formatação de página discutida rapidamente na Seção 24.3.2.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

O Project Management Institute (*Guide to the Project Management Body of Knowledge*, PMI, 2001) aborda todos os aspectos importantes do gerenciamento de projeto. Bechtold (*Essentials of Software Project Management*, 2d ed., Management Concepts, 2007), Wysocki (*Effective Software Project Management*, Wiley, 2006), Stellman e Greene (*Applied Software Project Management*, O'Reilly, 2005), e Berkun (*The Art of Project Management*, O'Reilly, 2005) ensinam práticas básicas e fornecem orientações detalhadas para todas as tarefas de gerenciamento de projeto de software. McConnell (*Professional Software Development*, Addison-Wesley, 2004) oferece informações pragmáticas para conseguir “cronogramas mais breves, produtos de melhor qualidade e projetos de melhor êxito”. Henry (*Software Project Management*, Addison-Wesley, 2003) oferece conselhos práticos que serão úteis para todos os gerentes de projeto.

Tom DeMarco e seus colegas (*Adrenaline Junkies and Template Zombies*, Dorset House, 2008) escreveram um tratado bastante esclarecedor sobre os padrões humanos encontrados em todos os projetos de software. Uma excelente série de quatro volumes escrita por Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996) introduz conceitos de pensamento e gerenciamento para sistemas básicos, explica como usar as medições eficazmente, e trata da “ação congruente”, a habilidade de estabelecer “a adaptação” entre as necessidades do gerente, as necessidades do pessoal técnico e as necessidades dos negócios. Ele apresenta informações úteis para gerentes iniciantes ou experientes. Futrell e seus colegas (*Quality Software Project Management*, Prentice-Hall, 2002) apresenta um volumoso tratado de gerenciamento de projeto. Brown e seus colegas (*Antipatterns in Project Management*, Wiley, 2000) discutem o que não fazer durante o gerenciamento de um projeto de software.

Brooks (*The Mythical Man-Month*, Anniversary Edition, Addison-Wesley, 1995) atualizou seu livro clássico para proporcionar nova visão sobre os aspectos de projeto e gerenciamento de software. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) apresenta excelente guia pragmático para aqueles que devem gerenciar projetos de software. Purba e Shah (*How to Manage a Successful Software Project*, 2d ed., Wiley, 2000) fornecem vários estudos de caso que indicam por que alguns projetos têm sucesso e outros não. Bennatan (*On Time Within Budget*, 3d ed., Wiley, 2000) dá dicas úteis e diretrizes para gerentes de projeto de software. Weigers (*Practical Project Initiation*, Microsoft Press, 2007) fornece orientações práticas para realizar de forma bem-sucedida um projeto de software desde o início.

Pode-se argumentar que o aspecto mais importante do gerenciamento de projeto de software seja o gerenciamento das pessoas. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) apresenta uma das melhores discussões sobre software desenvolvido atualmente. DeMarco e Lister [DeM98] produziram o livro definitivo sobre profissionais e projetos de software. Além disso, nos últimos anos foram publicadas também as seguintes obras que devem ser examinadas:

- Cantor, M., *Software Leadership: A Guide to Successful Software Development*, Addison-Wesley, 2001.
- Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones*, Prentice Hall, 1999.
- Constantine, L., *Peopleware Papers: Notes on the Human Side of Software*, Prentice Hall, 2001.
- Garton, C., e K. Wegryn, *Managing Without Walls*, McPress, 2006.
- Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process*, Addison-Wesley, 1997.
- Humphrey, W. S., *TSP-Coaching Development Teams*, Addison-Wesley, 2006.
- Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development*, McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments*, IEEE Computer Society, 1998.

Peters, L., *Getting Results from Software Development Teams*, Microsoft Press, 2008.

Whitehead, R., *Leading a Software Development Team*, Addison-Wesley, 2001.

Apesar de não se relacionarem especificamente com o mundo do software e muitas vezes apresentarem uma simplificação demasiada e uma ampla generalização, os *best-sellers* sobre “gerenciamento” de Kanter (*Confidence*, Three Rivers Press, 2006), Covy (*The 8th Habit*, Free Press, 2004), Bossidy (*Execution: The Discipline of Getting Things Done*, Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century*, Harper Business, 1999), Buckingham e Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently*, Simon e Schuster, 1999), e Christensen (*The Innovator's Dilemma*, Harvard Business School Press, 1997) enfatizam “novas regras” definidas por uma economia em rápida evolução. Títulos mais antigos como *Who Moved My Cheese?*, *The One-Minute Manager* e *In Search of Excellence* continuam a proporcionar visões valiosas que podem ajudá-lo a gerenciar pessoas e projetos mais eficazmente.

Uma ampla variedade de fontes de informação sobre gerenciamento de projeto de software pode ser encontrada na Internet. Uma lista atualizada das referências na Web, relevantes ao gerenciamento e projeto de software, pode ser obtida no site **www.mhhe.com/engcs/compSci/pressman/professional/olc/ser.htm**.

MÉTRICAS DE PROCESSO
E PROJETOCONCEITOS-
-CHAVE

eficiência na remoção de defeitos (defect removal efficiency – DRE)	595
medida	588
métricas	584
argumentos para.	596
baseado em LOC	591
estabelecendo um programa	599
linha de base	597
orientado a função.	589
orientado a objeto	591
orientado a caso de uso	592
orientado a tamanho	588
pontos de função	589
processo	584
produtividade	589
projeto	586
público e privado.	585
qualidade de software. .	594
WebApp	592

A medição nos permite obter o entendimento do processo e projeto, dando-nos um mecanismo para uma avaliação objetiva. Lord Kelvin disse certa vez:

Quando você pode medir o que você está falando e expressar em números, você sabe alguma coisa sobre aquilo; mas quando você não pode medir, quando você não pode expressar em números, o seu conhecimento é algo escasso e insatisfatório: pode ser o começo do conhecimento, mas você avançou muito pouco, em seu raciocínio, para o estágio de uma ciência.

A comunidade de engenharia de software levou a sério as palavras de Lord Kelvin. Mas não sem frustração e mais do que um pouco de controvérsia!

Medições podem ser aplicadas ao processo de software com a intenção de melhoria contínua. As medições podem ser usadas durante um projeto para ajudar nas estimativas, controle de qualidade, produtividade, e controle de projeto. Finalmente, a medição pode ser usada pelos engenheiros de software para ajudar a avaliar a qualidade dos artefatos e auxiliar na tomada de decisões táticas na medida em que o projeto evolui (Capítulo 23).

Dentro do contexto do processo de software e dos projetos que são executados usando o processo, uma equipe de software se preocupa primariamente com as métricas de produtividade e qualidade – medidas do “resultado” do desenvolvimento de software em função do esforço e do tempo aplicados e medidas da “adequação para uso” dos artefatos que são produzidos. Para fins de planejamento e estimativa, o seu interesse é histórico. Qual foi a produtividade do desenvolvimento de software nos projetos anteriores? Qual foi a qualidade do software produzido? Como a produtividade passada e os dados de qualidade podem ser extrapolados para o presente? Como isso pode ajudá-lo a fazer planos e estimativas mais precisos?

PANORAMA

O que é? As métricas de projeto e processo de software são medidas quantitativas que permitem que você tenha o discernimento sobre a eficácia

do processo de software e os projetos que são executados usando o processo como uma estrutura. São coletados dados básicos de qualidade e produtividade. Esses dados são então analisados, comparados com médias passadas, e avaliados para determinar se ocorreram melhorias de qualidade e produtividade. As métricas são usadas também para apontar áreas com problemas, de modo que as correções possam ser desenvolvidas e o processo de software melhorado.

Quem realiza? Métricas de software são analisadas e avaliadas por gerentes de software. As medidas muitas vezes são coletadas por engenheiros de software.

Por que é importante? Se você não medir, o julgamento só pode ser baseado em uma avaliação subjetiva. Com a medição, tendências (tanto boas

quanto ruins) podem ser detectadas, podem ser feitas melhores estimativas e melhorias significativas podem ser obtidas ao longo do tempo.

Quais são as etapas envolvidas? Comece definindo um conjunto limitado de medidas de processo, projeto e produto que sejam fáceis de coletar. Essas medidas são muitas vezes normalizadas usando-se métricas orientadas a tamanho ou função. O resultado é analisado e comparado com médias anteriores para projetos similares executados na organização. São avaliadas as tendências e são obtidas as conclusões.

Qual é o artefato? Uma série de métricas de software que proporcionam discernimento sobre o processo e possibilitam entender o projeto.

Como garantir que o trabalho foi realizado corretamente? Aplicando um esquema de medições consistente, embora simples que nunca deve ser usado para avaliar, recompensar, ou punir indivíduos por seu desempenho pessoal.

Em seu guia sobre medições de software, Park, Goethert, e Florac [Par96b] observam as razões por que medimos: (1) *caracterizar*, em um esforço para obter um conhecimento “de processos, produtos, recursos, e ambientes, e para estabelecer linhas de base para comparações com futuras avaliações”; (2) *avaliar* “para determinar o status com relação aos planos”; (3) *prever* “entendendo as relações entre os processos e produtos e criando modelos dessas relações”; e (4) *melhorar* “identificando etapas, causas raiz de problemas, ineficiências e outras oportunidades de melhoria da qualidade do produto e desempenho do processo”.

A medição é uma ferramenta de gerenciamento. Se for usada adequadamente, ela proporciona discernimento ao gerente de projeto. E, conseqüentemente, ela ajuda o gerente de projeto e a equipe de software a tomar decisões que levarão a um projeto bem-sucedido.

25.1 MÉTRICAS NO DOMÍNIO DE PROCESSO E PROJETO

PONTO-CHAVE

As métricas de processo têm impacto de longo prazo. Seu objetivo é melhorar o próprio processo. As métricas de projeto muitas vezes contribuem para o desenvolvimento de métricas de processo.

Métricas de processo são coletadas através de todos os projetos e sobre longos períodos de tempo. Sua finalidade é proporcionar uma série de indicadores de processo que levam à melhoria do processo de software no longo prazo. *Métricas de projeto* permitem ao gerente de projeto de software (1) avaliar o estado de um projeto em andamento, (2) rastrear os riscos em potencial, (3) descobrir áreas problemáticas antes que elas se tornem “críticas”, (4) ajustar o fluxo de trabalho ou tarefas, e (5) avaliar a habilidade da equipe de projeto para controlar a qualidade dos produtos finais de software.

Medidas que são coletadas por uma equipe de projeto e convertidas em métricas para uso durante um projeto podem também ser transmitidas para aqueles que têm responsabilidade pelo aperfeiçoamento do processo de software (Capítulo 30). Por essa razão, as mesmas métricas são usadas, muitas delas, tanto nos domínios de processo quanto de projeto.

25.1.1 Métricas de processo e aperfeiçoamento do processo de software

A única maneira racional de melhorar qualquer processo é medir atributos específicos do processo, desenvolver uma série de métricas significativas com base nesses atributos, e então usar as métricas para fornecer indicadores que levarão a uma estratégia de aperfeiçoamento (Capítulo 30). Mas antes de discutirmos as métricas de software e seu impacto sobre a melhoria do processo de software, é importante notar que o processo é apenas um item dentre uma série de “fatores controláveis na melhoria da qualidade do software e do desempenho organizacional” [Pau94].

De acordo com a Figura 25.1, o processo está no centro do triângulo que conecta três fatores que possuem uma profunda influência sobre a qualidade do software e desempenho organizacional. Já foi mostrado [Boe81] que a habilidade e a motivação das pessoas representam o fator individual mais influente na qualidade e no desempenho. A complexidade do produto pode ter um impacto substancial sobre a qualidade e desempenho da equipe. A tecnologia, (isto é, os métodos e ferramentas de engenharia de software) que preenche o processo tem também um impacto.

Além disso, o triângulo do processo encontra-se dentro de um círculo de condições ambientais que inclui o ambiente de desenvolvimento (por exemplo, ferramentas de software integradas), condições de negócios (por exemplo, prazos de entrega, regras de negócio) e características do cliente (por exemplo, facilidade de comunicação e colaboração).

Você só pode medir a eficácia de um processo de software indiretamente. Isto é, você cria uma série de métricas baseadas nos resultados que podem ser obtidos do processo. Os resultados incluem medidas de erros descobertos antes da entrega do software, defeitos relatados pelos usuários finais, produtos acabados fornecidos (produtividade), esforço humano gasto, tempo despendido, conformidade com o cronograma, e outras medidas. Você pode também obter métricas de processo medindo as características de tarefas específicas de engenharia de software. Por exemplo, você pode medir o esforço e o tempo despendidos executando as atividades genéricas de engenharia de software descritas no Capítulo 2.

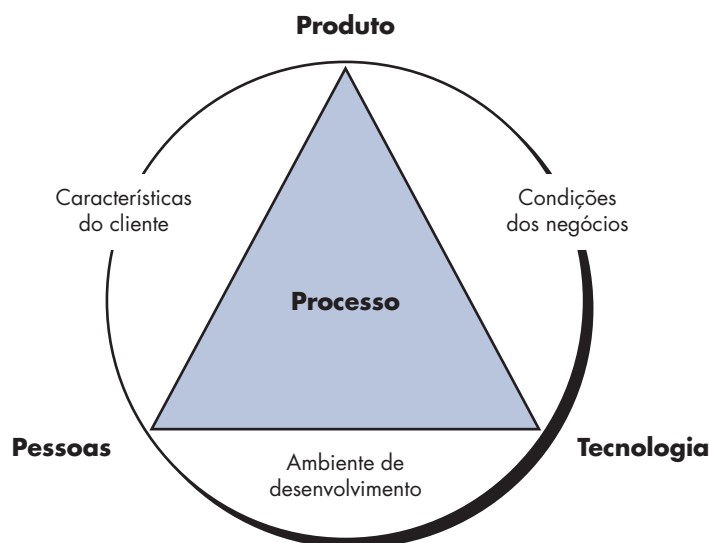
PONTO-CHAVE

A habilidade e motivação dos profissionais que fazem o trabalho são os fatores mais importantes que influenciam na qualidade do software.

FIGURA 25.1

Determinantes para a qualidade do software e eficácia organizacional

Fonte: Adaptado de (Pau94)



“As métricas de software permitem que você saiba quando deve rir e quando deve chorar.”

Tom Gilb

Qual é a diferença entre usos público e privado para as métricas de software?

Que diretrizes devem ser aplicadas quando coletamos métricas de software?

Grady [Gra92] afirma que há usos “privados e públicos” para diferentes tipos de dados de processo. Como é natural que os engenheiros de software individualmente podem ser sensíveis ao uso de métricas coletadas individualmente, esses dados devem ser privados e servir apenas como um indicador individual. Exemplos de *métricas privadas* incluem taxas de defeito (por indivíduo), taxas de defeito (por componente) e erros encontrados durante o desenvolvimento.

A filosofia de “dados privados do processo” combina bem com a abordagem Personal Software Process (Capítulo 2) proposta por Humphrey [Hum97]. Ele reconhece que o aperfeiçoamento do processo de software pode e deve começar em nível individual. Dados privados do processo podem servir como um motivador importante quando você trabalha para melhorar a sua abordagem de engenharia de software.

Algumas métricas de processo são privadas para a equipe de projeto de software, mas são públicas para todos os membros da equipe. Exemplos incluem defeitos relatados para funções principais do software (que foram desenvolvidas por vários profissionais), erros encontrados durante revisões técnicas, e linhas de código ou pontos de função por componente ou função.¹ A equipe examina esses dados para descobrir indicadores que podem melhorar o desempenho da equipe.

Métricas públicas geralmente assimilam informações que originalmente eram privadas aos indivíduos e equipes. Taxas de defeito em nível de projeto (absolutamente não atribuídas aos indivíduos), esforço, prazos agendados, e dados relacionados são coletados e avaliados tentando descobrir indicadores que podem melhorar o desempenho do processo organizacional.

Métricas de processo de software podem produzir benefícios significativos quando uma organização trabalha para melhorar seu nível geral de maturidade de processo. No entanto, assim como todas as métricas, essas podem ser mal utilizadas, criando mais problemas do que elas podem resolver. Grady [Gra92] sugere uma “etiqueta de métricas de software” apropriada para os gerentes e para os profissionais quando instituem um programa de métricas de processo:

- Use bom senso e sensibilidade organizacional ao interpretar dados de métricas.
- Forneça regularmente feedback aos indivíduos e equipes que coletam medidas e métricas.
- Não use métricas para avaliar indivíduos.
- Trabalhe com profissionais e equipes para definir objetivos claros e métricas que serão usadas para alcançá-las.

¹ Métricas de linhas de código e pontos de função são discutidas nas Seções 25.2.1 e 25.2.2.

- Nunca use métricas para ameaçar indivíduos ou equipes.
- Dados de métricas que indicam uma área com problema não deverão ser considerados “negativos”. Esses dados são simplesmente um indicador para melhoria do processo.
- Não seja obsessivo sobre uma única métrica excluindo as outras métricas importantes.

À medida em que uma organização se sente mais à vontade, coletando e usando métricas de processo, a derivação de indicadores simples dá margem a uma abordagem mais rigorosa chamada *melhoria estatística de processo de software* (*statistical software process improvement – SSPI*). Essencialmente, a SSPI usa a análise de falhas de software para coletar informações sobre todos os erros e defeitos² encontrados quando uma aplicação, sistema, ou produto é desenvolvido e usado.

25.1.2 Métricas de projeto

Diferentemente das métricas de processo de software que são usadas para fins estratégicos, as medidas de projeto de software são táticas. Isto é, métricas de projeto e os indicadores derivados delas são usados por um gerente de projeto e uma equipe de software para adaptar o fluxo de trabalho do projeto e as atividades técnicas.

A primeira aplicação das métricas de projeto na maioria dos projetos de software ocorre durante as estimativas. Métricas coletadas de projetos passados são usadas como base a partir da qual são feitas as estimativas de esforços e tempo para o trabalho atual de software. Na medida em que um projeto progride, medidas de esforço e tempo despendidos são comparadas com as estimativas originais (e com o cronograma do projeto). O gerente de projeto usa esses dados para monitorar e controlar o progresso.

Quando o trabalho técnico começa, outras métricas de projeto começam a ter significado. São medidas as taxas de produção representadas em termos de modelos criados, revisões, pontos de função e linhas de código-fonte fornecidas. Além disso, são rastreados os erros descobertos durante cada tarefa de engenharia de software. Na medida em que o software evolui dos requisitos para o projeto, métricas técnicas (Capítulo 23) são coletadas para avaliar a qualidade do projeto e fornecer indicadores que terão influência na abordagem adotada para geração de código e teste.

O objetivo das métricas de projeto é duplo. Primeiro, essas métricas são usadas para minimizar o cronograma de desenvolvimento fazendo os ajustes necessários para evitar atrasos e mitigar problemas e riscos em potencial. Segundo, as métricas de projeto são usadas para avaliar a qualidade do produto de forma contínua e, quando necessário, modificar a abordagem técnica para melhorar a qualidade.

À medida que a qualidade melhora, os defeitos são minimizados, e à medida que a contagem de defeitos diminui, a quantidade de retrabalho necessário durante o projeto também é reduzida. Isso leva a uma redução no custo total do projeto.

 **Como devemos usar as métricas durante o próprio projeto?**

25.2 MEDIDAS DE SOFTWARE

“Nem tudo o que pode ser contado importa, e nem tudo o que importa pode ser contado.”

Albert Einstein

No Capítulo 23, observamos que as medidas no mundo físico podem ser classificadas de duas maneiras: medidas diretas (por exemplo, o comprimento de um parafuso) e medidas indiretas (por exemplo, a “qualidade” dos parafusos produzidos, medida contando os rejeitos). As métricas de software podem ser classificadas de maneira similar.

Medidas diretas do processo de software incluem custos e trabalho aplicado. Medidas diretas do produto incluem linhas de código (lines of code - LOC) produzidas, velocidade de execução, tamanho de memória e defeitos relatados durante um determinado período de tempo. Medidas

² Neste livro, um *erro* é definido como alguma falha em um artefato que é descoberta *antes* que o software seja fornecido ao usuário final. Um *defeito* é uma falha descoberta *depois* que o software é fornecido ao usuário final. Devemos destacar que muitas pessoas não fazem essa distinção.

CASASEGURA



Estabelecendo uma abordagem de métricas

A cena: Escritório de Doug Miller, quando o projeto de software CasaSegura está para começar.

Os atores: Doug Miller (gerente da equipe de engenharia de software CasaSegura) e Vinod Raman e Jamie Lazar, membros da equipe de engenharia de artefatos.

Conversa:

Doug: Antes de começarmos a trabalhar neste projeto, eu gostaria que vocês definissem e coletassem uma série de métricas simples. Para começar, vocês terão que definir nossas metas.

Vinod (com cara de contrariado): Nós nunca fizemos isso antes, e...

Jamie (interrompendo): E com base na administração do tempo sobre a qual temos conversado, nós nunca teremos tempo. Afinal, essas métricas são mesmo boas?

Doug (erguendo a mão para interromper a discussão): Calma, respirem fundo rapazes. O fato de nunca termos feito isso antes é a melhor razão para começar a fazer agora, e o trabalho com as métricas das quais eu estava falando não deve tomar muito tempo... Na verdade, elas podem até nos poupar tempo.

Vinod: Como?

Doug: Olhe, faremos muito mais trabalho interno de engenharia de software à medida que nossos produtos se tornarem mais

inteligentes, mais voltados à Web, tudo isso... E precisaremos entender o processo que usamos para criar o software... E melhorá-lo para criarmos um software melhor. A única maneira de fazer isso é por meio das medições.

Jamie: Mas estamos sendo pressionados no prazo, Doug. Não sou a favor de gerar mais papel... Precisamos de tempo para fazer nosso trabalho, não para coletar dados.

Doug (calmamente): Jamie, o trabalho de um engenheiro envolve coleta de dados, avaliação desses dados, e o uso do resultado para melhorar o produto e o processo. Estou errado?

Jamie: Não, mas...

Doug: E que tal se mantivermos o número de medidas que coletamos em não mais de cinco ou seis e focarmos a qualidade?

Vinod: Ninguém pode argumentar contra a alta qualidade...

Jamie: Certo... Mas, eu não sei. Eu ainda acho que não é necessário.

Doug: Peço que sejam tolerantes comigo neste ponto. O que vocês sabem sobre métricas?

Jamie (olhando para Vinod): Não muito.

Doug: Aqui estão algumas referências da Web... Dediquem algumas horas para se informar.

Jamie (sorrindo): Eu achei que você tinha falado que isso não tomaria muito tempo.

Doug: O tempo que você gasta aprendendo nunca é perdido... Façam isso e então estabeleceremos nossas metas, façam algumas perguntas, e definam as métricas que precisamos coletar.

indiretas do produto incluem funcionalidade, qualidade, complexidade, eficiência, confiabilidade, manutenibilidade, e muitas outras “ades” que são discutidas no Capítulo 14.

O custo e trabalho requerido para criar o software, o número de linhas de código produzidas e outras medidas diretas são relativamente fáceis de coletar, desde que sejam estabelecidas antecipadamente convenções para as medições. No entanto, a qualidade e a funcionalidade do software ou sua eficiência ou manutenibilidade são mais difíceis de avaliar e podem ser medidas somente de forma indireta.

Nós dividimos o domínio de métricas de software em processo, projeto, e métricas de produto e observamos que as métricas de produto que são privadas a um indivíduo são muitas vezes combinadas para desenvolver métricas de projeto que são públicas para a equipe de software. Métricas de projeto são então consolidadas para criar métricas de processo que são públicas à organização de software como um todo. Mas como uma organização combina métricas que vieram de diferentes indivíduos ou projetos?

Para ilustrar, considere um exemplo simples. Indivíduos em duas equipes de projeto diferentes registram e classificam todos os erros que eles encontram durante o processo de software. As medidas individuais são então combinadas para criar medidas de equipe. A Equipe A encontrou 342 erros durante o processo de software antes da entrega. A Equipe B encontrou 184 erros. Sendo iguais todas as outras coisas, qual equipe é mais eficaz na descoberta de erros em todo o processo? Como você não sabe qual é o tamanho ou complexidade dos projetos, você não pode responder a essa pergunta. No entanto, se as medidas forem normalizadas, é possível criar métricas de software que possibilitam a comparação com médias organizacionais mais amplas.



Como muitos fatores afetam o trabalho de software, não use métricas para comparar indivíduos ou equipes.

FIGURA 25.2

Métricas orientadas a tamanho

Projeto	Linhas de código	Esforço	\$(000)	Pág. doc.	Erros	Defeitos	Pessoas
alfa	12.100	24	168	365	134	29	3
beta	27.200	62	440	1224	321	86	5
gama	20.200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

25.2.1 Métricas orientadas a tamanho

Métricas de software orientadas a tamanho são criadas normalizando-se as medidas de qualidade e/ou produtividade considerando o *tamanho* do software que foi produzido. Se uma organização de software mantém registros simples, pode ser criada uma tabela de medidas orientadas a tamanho, como aquela mostrada na Figura 25.2. A tabela lista cada projeto de desenvolvimento de software que foi completado durante alguns anos passados e medidas correspondentes para aquele projeto. Olhando a linha da tabela (Figura 25.2) para o projeto alfa: 12.100 linhas de código foram criadas com 24 pessoas-mês de trabalho a um custo de \$168.000. Deve-se observar que todo o trabalho e custo registrados na tabela representa todas as atividades de engenharia de software (análise, projeto, código e teste), não apenas codificação. Outras informações para o projeto alfa indicam que foram criadas 365 páginas de documentação, foram registrados 134 erros antes da entrega do software e foram encontrados 29 defeitos após a entrega para o cliente durante o primeiro ano de operação. Três pessoas trabalharam no desenvolvimento do software para o projeto alfa.

Para desenvolver métricas que podem ser assimiladas com métricas similares de outros projetos, você pode escolher número de linhas de código como um valor de normalização. A partir dos dados rudimentares contidos na tabela, pode ser desenvolvido um conjunto de métricas simples orientadas a tamanho para cada projeto:

- Erros por KLOC (mil linhas de código)
- Defeitos por KLOC
- \$ por KLOC
- Páginas de documentação por KLOC

Além disso, podem ser computadas outras métricas interessantes:

- Erros por pessoa-mês
- KLOC por pessoa-mês
- \$ por página de documentação

Métricas orientadas por tamanho não são aceitas universalmente como a melhor maneira de medir os processos de software. A maior parte da controvérsia gira em torno do uso de linhas de código como medida principal. Os proponentes da medida LOC (linhas de código) argumentam que LOC é um “item” de todos os projetos de desenvolvimento de software que pode ser facilmente contado, que muitos modelos de estimativa de software existentes usam LOC ou KLOC como dado de entrada principal, e que já existe uma grande quantidade de literatura e

PONTO-CHAVE

Métricas orientadas a tamanho são amplamente usadas, mas continua o debate sobre sua validade e aplicabilidade.

dados baseados em LOC. Por outro lado, os oponentes argumentam que as medidas LOC são dependentes da linguagem de programação, que quando é considerada a produtividade, elas penalizam programas bem projetados, mas menores; que elas não podem facilmente acomodar linguagens não procedurais; e que seu uso nas estimativas requer um nível de detalhe que pode ser difícil de alcançar (isto é, o planejador deve estimar a LOC para ser produzida bem antes que a análise e o projeto estejam completados).

25.2.2 Métricas orientadas a função

Métricas de software orientadas a função usam uma medida da funcionalidade fornecida pela aplicação como um valor de normalização. A métrica orientada a função mais amplamente usada é a pontos de função (function point – FP). O cálculo de pontos de função é baseada nas características de domínio de informação e complexidade do software. O mecanismo de cálculo de FP foi discutido no Capítulo 23.³

O ponto de função, assim como a medida LOC, é controverso. Os proponentes argumentam que essa função é independente da linguagem de programação, tornando-a ideal para aplicações que usam linguagens convencionais e não procedurais, e que é baseada em dados que têm maior probabilidade de ser conhecidos na evolução de um projeto, tornando a FP mais atraente como abordagem de estimativa. Os oponentes argumentam que o método requer um pouco de “jeitinho”, porque o cálculo é baseado em dados subjetivos ao invés de objetivos, que as contagens do domínio de informações (e outras dimensões) podem ser difíceis de coletar após o fato, e que a FP não tem um significado físico direto – é apenas um número.

25.2.3 Reconciliando métricas LOC e FP

A relação entre linhas de código e pontos de função depende da linguagem de programação que é usada para implementar o software e a qualidade do projeto. Muitos estudos já tentaram relacionar medidas FP e LOC. A tabela a seguir⁴ [QSM02] fornece estimativas aproximadas da média do número de linhas de código necessárias para criar um ponto de função em várias linguagens de programação:

Uma revisão desses dados indica que uma LOC de C++ fornece aproximadamente 2,4 vezes a “funcionalidade” (em média) de uma LOC de C. Além disso, uma LOC de Smalltalk fornece pelo menos quatro vezes a funcionalidade de uma LOC para uma linguagem de programação convencional como Ada, COBOL, ou C. Usando as informações da tabela, é possível “retroagir” [Jon98] o software existente para estimar o número de pontos de função, uma vez conhecido o número total de instruções da linguagem de programação.

Medidas de LOC e FP muitas vezes são usadas para derivar métricas de produtividade. Isso leva invariavelmente a um debate sobre o uso de tais dados. Deveria o valor LOC/pessoa-mês (ou FP/pessoa-mês) de um grupo ser comparado com dados similares de outro grupo? Os gerentes deveriam avaliar o desempenho dos indivíduos usando essas métricas? A resposta a essas questões é um enfático “Não!” A razão para essa resposta é que muitos fatores influenciam a produtividade, gerando comparações do tipo “maças com laranjas”, que podem facilmente ser mal interpretadas.

Descobriu-se que métricas baseadas em pontos de função e LOC são indicadores relativamente precisos do trabalho e custo do desenvolvimento de software. No entanto, para usar LOC e FP para estimativas (Capítulo 26), deve ser estabelecida uma referência histórica de informações.

Dentro do contexto de métricas de processo e projeto, você deve se preocupar em primeiro lugar com a produtividade e a qualidade – medidas de “resultado” de desenvolvimento de software em função do esforço e tempo aplicados e as medidas da “adequação para uso” dos produtos que são criados. Para fins de melhoria de processo e planejamento de projeto, seu interesse é histórico. Qual foi a produtividade do desenvolvimento de software nos projetos passados?

3 Veja na Seção 23.2.1 uma discussão detalhada da computação FP.

4 Usado com permissão de Quantitative Software Management (www.qsm.com), copyright 2002.

Linguagem de Programação	LOC por Ponto de Função			
	Média	Mediana	Baixa	Alta
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool:Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Excel 47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool:Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—
JavaScript	58	63	42	75
JCL	91	123	26	150
JSP	59	—	—	—
Lotus Notes	21	22	15	25
Mantis	71	27	22	250
Mapper	118	81	16	245
Natural	60	52	22	141
Oracle	30	35	4	217
PeopleSoft	33	32	30	40
Perl	60	—	—	—
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
REXX	67	—	—	—
RPG II/III	61	49	24	155
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
VBScript 36	34	27	50	—
Visual Basic	47	42	16	158

Qual foi a qualidade do software produzido? Como os dados de produtividade e qualidade do passado podem ser extrapolados para o presente? Como isso pode nos ajudar a melhorar o processo e planejar novos projetos mais precisamente?

25.2.4 Métricas orientadas a objeto

Métricas de projeto de software convencional (LOC ou FP) podem ser usadas para estimar projetos de software orientados a objeto. No entanto, essas métricas não fornecem granularidade suficiente para os ajustes de cronograma e esforço que são necessários na medida em que você passa por iterações por meio de um processo evolucionário ou incremental. Lorenz e Kidd [Lor94] sugerem o seguinte conjunto de métricas para projetos orientados a objeto:

Número de scripts de cenário (*Number of scenario scripts*). Um script de cenário (análogo aos casos de uso discutidos na Parte 2 deste livro) é uma sequência detalhada de passos que descrevem a interação entre o usuário e a aplicação. Cada script é organizado em trios da forma

{**iniciador**, *ação*, **participante**}

onde **iniciador** é o objeto que solicita algum serviço (que inicia uma mensagem), *ação* é o resultado da solicitação, e **participante** é o objeto servidor que satisfaz a solicitação. O número de scripts de cenário está correlacionado diretamente com o tamanho da aplicação e com o número de casos de testes que devem ser desenvolvidos para exercitar o sistema depois que ele é construído.



Não é raro ocorrer que scripts de múltiplos cenários mencionem a mesma funcionalidade ou objetos dados. Portanto, tenha cuidado ao usar contagem de scripts. Muitos scripts podem às vezes ser reduzidos a uma única classe ou conjunto de código.

Número de classes-chave (*Number of key classes*). *Classes-chave* (*Key classes*) são os “componentes altamente independentes” [Lor94] que são definidos logo no início em análise orientada a objeto (Capítulo 6).⁵ Como as classes-chave são essenciais ao domínio do problema, a quantidade dessas classes é uma indicação da quantidade de esforço necessário para desenvolver o software e também uma indicação do potencial de reutilização a ser aplicado durante o desenvolvimento do sistema.

Número de classes de apoio (*Number of support classes*). *Classes de apoio* (*Support classes*) são necessárias para implementar o sistema, mas não estão imediatamente relacionadas com o domínio do problema. Como exemplos podemos citar as classes de interface de usuário (GUI), classes de acesso e manipulação de bases de dados, e classes de cálculo. Além disso, podem ser desenvolvidas classes de apoio para cada uma das classes-chave. As *classes de apoio* são definidas iterativamente durante um processo evolucionário. O número de classes de apoio é uma indicação da quantidade de esforço necessário para desenvolver o software e também uma indicação do potencial de reutilização a ser aplicado durante o desenvolvimento do sistema.



As classes podem variar em tamanho e complexidade. Portanto, pense em classificar as contagens de classes por tamanho e complexidade.

Número médio de classes de apoio para cada classe-chave (*Average number of support classes per key class*). Em geral, as classes-chave são conhecidas logo no início do projeto. As classes de apoio são definidas durante o projeto. Se o número médio de classes de apoio para cada classe-chave fosse conhecido para um dado domínio de problema, a estimativa (baseada no número total de classes) seria muito simplificada. Lorenz e Kidd sugerem que as aplicações com uma GUI tenham de duas a três vezes a quantidade de classes suporte como classes-chave. Aplicações não GUI têm de uma a duas vezes a quantidade de classes suporte como classes-chave.

Número de subsistemas (*Number of subsystems*). Um *subsistema* é uma agregação de classes que apoia uma função que é visível ao usuário final de um sistema. Uma vez identificados os subsistemas, é mais fácil elaborar um cronograma razoável no qual o trabalho nos subsistemas é dividido entre o pessoal de projeto.

5 Nós nos referimos às classes-chave como classes de análise na Parte 2 deste livro.

Para serem usadas eficazmente em um ambiente de engenharia de software orientado a objeto, métricas similares às aquelas mencionadas acima deverão ser coletadas juntamente com as medidas de projeto, tais como o esforço gasto, erros e defeitos descobertos, e modelos ou páginas de documentação produzidas. Na medida em que a base de dados cresce (após completar um grupo de projetos), as relações entre as medidas orientadas a objeto e as medidas de projeto fornecerão métricas que podem ajudar nas estimativas do projeto.

25.2.5 Métricas orientadas a casos de uso

Os casos de uso⁶ são amplamente usados como método para descrever requisitos no nível dos clientes ou domínio de negócio que sugerem características e funções de software. Seria considerado razoável usar o caso de uso como uma medida de normalização similar a LOC ou FP. Assim como a FP, o caso de uso é definido no início no processo de software, permitindo que ele seja usado para estimativas antes de iniciar atividades significativas de modelagem e construção. Os casos de uso descrevem (indiretamente, pelo menos) funções e características visíveis ao usuário que são requisitos básicos para um sistema. O caso de uso é independente da linguagem de programação. Além disso, o número de casos de uso é diretamente proporcional ao tamanho do aplicativo em LOC e ao número de casos de testes que terão de ser projetados para exercitar completamente o aplicativo.

Como os casos de uso podem ser criados em níveis muito diferentes de abstração, não há um “tamanho” padrão para um caso de uso. Sem uma medida padronizada do que é um caso de uso, sua aplicação como medida de normalização (por exemplo, esforço gasto por cada caso de uso) é suspeita.

Os pesquisadores têm sugerido os *pontos de casos de uso* (UCPs) como um mecanismo para estimar trabalho de projeto e outras características. O UCP é uma função do número de atores e transações deduzidas pelos modelos de casos de uso e é análogo ao FP em alguns aspectos. Se você tiver mais interesse, veja [Cle06].

25.2.6 Métricas de projeto WebApp

O objetivo de todos os projetos WebApp é fornecer uma combinação de conteúdo e funcionalidade para o usuário final. Medidas e métricas usadas para projetos tradicionais de engenharia de software são difíceis de traduzir diretamente para WebApps. No entanto, é possível desenvolver uma base de dados que permite acesso a medidas internas de produtividade e qualidade obtidas em uma série de projetos. Entre as medidas que podem ser coletadas estão:

Número de páginas Web estáticas. Páginas Web com conteúdo estático (isto é, o usuário final não tem controle sobre o conteúdo mostrado na página) são as mais comuns de todas as características WebApp. Essas páginas representam baixa complexidade relativa e geralmente requerem menos esforços para ser criadas do que as páginas dinâmicas. Essa medida fornece uma indicação do tamanho global da aplicação e do esforço necessário para desenvolvê-la.

Número de páginas Web dinâmicas. Páginas Web com conteúdo dinâmico (isto é, as ações do usuário final ou outros fatores externos resultam em conteúdo personalizado apresentado na página) são essenciais em todos os aplicativos e-commerce, recursos de pesquisa, aplicações financeiras e muitas outras categorias WebApp. Essas páginas representam uma complexidade relativa maior e requerem mais esforço para ser construídas do que as páginas estáticas. Essa medida fornece uma indicação do tamanho geral do aplicativo e o esforço necessário para desenvolvê-lo.

Número de links de páginas internos. Links de páginas internos são ponteiros que fornecem um hyperlink para alguma outra página Web dentro do WebApp. Essa medida fornece uma indicação do grau de acoplamento arquitetônico dentro do WebApp. Na medida em que o número de links de páginas aumenta, aumenta também o esforço gasto no projeto navegacional e na construção.

6 Casos de uso são apresentados nos Capítulos 5 e 6.

Número de objetos dados persistentes. Um ou mais objetos dados persistentes (por exemplo, uma base de dados ou um arquivo de dados) pode ser acessado por uma WebApp. Na medida em que cresce o número de objetos dados persistentes, a complexidade do WebApp também cresce e o esforço para implementá-la aumenta proporcionalmente.

Número de sistemas externos interfaceados. Os WebApps muitas vezes devem se interfacear com aplicações de negócio de “retaguarda”. Na medida em que cresce o requisito para interfaceamento, a complexidade do sistema e esforço de desenvolvimento também aumenta.

Número de objetos de conteúdo estático. Objetos de conteúdo estático abrangem objetos estáticos baseados em texto, objetos gráficos, vídeo, animação e informações de áudio que são incorporadas dentro do WebApp. Objetos de conteúdo múltiplo podem aparecer em uma única página Web.

Número de objetos de conteúdo dinâmico. Objetos de conteúdo dinâmico são gerados baseados nas ações do usuário final e abrangem informações de texto geradas internamente, informações gráficas, vídeo, animação, e áudio, que são incorporadas dentro do WebApp. Objetos de conteúdo múltiplo podem aparecer em uma única página Web.

Número de funções executáveis. Uma função executável (por exemplo, um script ou applet) proporciona algum serviço de cálculo ao usuário final. Na medida em que o número de funções executáveis aumenta, os esforços de modelagem e construção também aumentam.

Cada uma das medidas que acabamos de mencionar pode ser determinada em um estágio relativamente antecipado. Por exemplo, você pode definir uma métrica que reflete o grau de personalização de usuário final requerida para o WebApp e correlacionar isso com o esforço gasto no projeto e/ou os erros descobertos na medida em que são feitas as revisões e testes. Para isso, você define

$$N_{sp} = \text{número de Páginas Web Estáticas}$$

$$N_{dp} = \text{número de Páginas Web Dinâmicas}$$

Então,

$$\text{Índice de personalização, } C = \frac{N_{dp}}{N_{dp} + N_{sp}}$$

O valor de C varia de 0 a 1. Na medida em que C se torna maior, o nível de personalização do WebApp se torna um aspecto técnico significativo.

Métricas WebApp similares podem ser calculadas e correlacionadas com medidas de projeto, tais como esforço gasto, erros e defeitos descobertos e modelos ou páginas de documentação produzidos. Na medida em que a base de dados cresce (após completar certo número de projetos), relações entre as medidas WebApp e medidas de projeto proporcionarão indicadores que podem ajudar na estimativa do projeto.

FERRAMENTAS DO SOFTWARE



Métricas de projeto e processo

Objetivo: auxiliar na definição, coleta, avaliação e relatórios de medidas e métricas de software.

Mecanismos: cada ferramenta varia em sua aplicação, mas todas elas fornecem mecanismos para coletar e avaliar dados que levam à computação de métricas de software.

Ferramentas Representativas:⁷

Function Point WORKBENCH, desenvolvida pela Charismatek (www.charismatek.com.au), oferece uma ampla gama de métricas orientadas para FP.

MetricCenter, desenvolvida pela Distributive Software (www.distributive.com), suporta coleta automática de dados, análise, formatação de gráficos, geração de relatórios e outras tarefas de medição.

PSM Insight, desenvolvida pela Practical Software and Systems Measurement (www.psmisc.com), ajuda na criação e subsequente análise de uma base de dados de medições de projeto.

SLIM tool set, desenvolvida pela QSM (www.qsm.com), contém um conjunto abrangente de métricas e ferramentas de estimativas.

SPR tool set, desenvolvida pela Software Productivity Research (www.spr.com), oferece uma coleção abrangente de ferramentas orientadas a FP.

TychoMetrics, desenvolvida pela Predicate Logic, Inc. (www.predicate.com), é um conjunto de ferramentas para coleta e relato de métricas de gerenciamento.

⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

25.3 MÉTRICAS PARA QUALIDADE DE SOFTWARE



Software é uma entidade complexa. Portanto, deve-se esperar que ocorram erros na medida em que o produto é desenvolvido. As métricas de processo são destinadas a melhorar o processo de software para que os erros sejam descobertos pela maneira mais eficiente.

O principal objetivo da engenharia de software é produzir um sistema, aplicação ou produto, de alta qualidade, dentro de um prazo que satisfaça as necessidades do mercado. Para atingir esse objetivo, devem-se aplicar métodos eficazes, combinados com modernas ferramentas dentro do contexto de um processo de software bem desenvolvido. Além disso, um bom engenheiro de software (e bons gerentes de engenharia de software) devem medir se a alta qualidade será obtida.

A qualidade de um sistema, aplicação, ou produto, é apenas tão boa quanto os requisitos que descrevem o problema, o projeto que modela a solução, o código que leva ao programa executável, e os testes que exercitam o software para descobrir os erros. Você pode usar medidas para avaliar a qualidade dos requisitos e modelos de projeto, o código-fonte, e os casos de testes que foram criados enquanto o software é desenvolvido. Para conseguir essa avaliação em tempo real, você aplica métricas de produto (Capítulo 23) para avaliar a qualidade dos artefatos de software de maneira objetiva, e não subjetiva.

Um gerente de projeto deve também avaliar a qualidade enquanto o projeto avança. Métricas privadas coletadas por engenheiros de software individuais são combinadas para fornecer os resultados no nível de projeto. Embora muitas medidas de qualidade possam ser coletadas, a principal tendência no nível de projeto é medir erros e defeitos. Métricas derivadas dessas medidas proporcionam uma indicação da efetividade da garantia de qualidade de software individual e de grupo e das atividades de controle.

Métricas como erros de produto por ponto de função, erros descobertos por horas de revisão, e erros descobertos por horas de teste proporcionam informações sobre a eficácia de cada uma das atividades sugeridas pela métrica. Os dados sobre os erros também podem ser usados para calcular a *eficiência de remoção de defeitos* (*defect removal efficiency* – DRE) para cada atividade da estrutura de processo. A DRE é discutida na Seção 25.3.3.

25.3.1 Medição da qualidade

Embora existam muitas medidas de qualidade de software,⁸ a correção, a manutenibilidade, integridade e usabilidade fornecem indicadores úteis para a equipe de projeto. Gilb [Gil88] sugere definições e medidas para cada uma delas.

WebRef

Uma excelente fonte de informações sobre qualidade de software e tópicos relacionados (incluindo métricas) pode ser encontrada em www.qualityworld.com.

Correção. Um programa deve operar corretamente ou terá pouca utilidade para seus usuários. A correção é o grau com o qual o software executa sua função. A medida mais comum da exatidão é o número de defeitos por KLOC, onde um defeito é definido como uma ocorrência de falta de conformidade com os requisitos. Ao considerar a qualidade geral de um artefato, os defeitos são aqueles problemas relatados por um usuário do programa depois que o programa foi liberado para uso geral. Para fins de avaliação de qualidade, os defeitos são contados durante um período de tempo padrão, em geral um ano.

Manutenibilidade. A manutenção do software e o suporte exigem maiores esforços do que qualquer outra atividade de engenharia. A manutenibilidade é a facilidade com a qual um programa pode ser corrigido se for encontrado um erro, adaptado se o ambiente mudar, ou melhorado se o cliente desejar uma alteração nos requisitos. Não há uma maneira de medir a manutenibilidade diretamente, portanto, é preciso usar medidas indiretas. Uma métrica simples orientada por tempo é o *tempo médio de alteração* (*mean-time-to-change* – MTTC), o tempo necessário para analisar a solicitação de alteração, projetar uma modificação apropriada, implementar a alteração, testá-la, e distribuí-la a alteração para todos os usuários. Na média, os programas manuteníveis terão um MTTC mais baixo (para tipos equivalentes de alteração) do que programas que não são manuteníveis.

⁸ Uma discussão detalhada dos fatores que influenciam na qualidade do software e as métricas que podem ser usadas para avaliar a qualidade do software foi apresentada no Capítulo 23.

Integridade. A integridade do software vem se tornando cada vez mais importante na era dos terroristas e hackers cibernéticos. Esse atributo mede a habilidade de um sistema em resistir aos ataques (tanto acidentais quanto intencionais) à sua segurança. Os ataques podem ser feitos em todos os três componentes do software: programas, dados e documentos.

Para medir a integridade, devem ser definidos dois atributos adicionais: ameaça e segurança. *Ameaça* é a probabilidade (que pode ser estimada ou derivada de evidência empírica) de que um ataque de um tipo específico ocorrerá em um dado intervalo de tempo. *Segurança* é a probabilidade (que pode ser estimada ou derivada de evidência empírica) de que um ataque de um tipo específico será repellido. A integridade de um sistema pode então ser definida como:

$$\text{Integridade} = \Sigma [1 - (\text{ameaça} \times (1 - \text{segurança}))]$$

Por exemplo, se a ameaça (probabilidade de que um ataque possa ocorrer) for 0,25 e a segurança (a possibilidade de repelir o ataque) for 0,95, a integridade do sistema é 0,99 (muito alta). Por outro lado, se a probabilidade de ameaça for 0,50 e a possibilidade de repelir um ataque for de apenas 0,25, a integridade do sistema é 0,63 (muito baixa e inaceitável).

Usabilidade. Se um programa não for fácil de usar, muitas vezes ele está destinado ao fracasso, mesmo que as funções que ele executa sejam valiosas. A usabilidade é uma tentativa de quantificar a facilidade de uso e pode ser medida em termos das características apresentadas no Capítulo 11.

Os quatro fatores que acabamos de descrever são apenas alguns exemplos daqueles que foram propostos como medidas para a qualidade do software. O Capítulo 23 considera este tópico com mais detalhes.

25.3.2 Eficiência na remoção de defeitos

Uma métrica de qualidade que traz benefícios tanto para o projeto quanto para o processo é a eficiência na remoção de defeitos (*defect removal efficiency* – DRE). Essencialmente, a DRE é uma medida da habilidade de filtragem das ações de garantia de qualidade e controle quando elas são aplicadas através de todas as atividades da estrutura de processo.

Quando considerada para um projeto como um todo, a DRE é definida da seguinte maneira:

$$\text{DRE} = \frac{E}{E + D}$$

onde E é o número de erros encontrados antes que o software seja fornecido ao usuário final e D é o número de defeitos depois que o software é entregue.

O valor ideal para DRE é 1. Isto é, nenhum defeito é encontrado no software. Realisticamente, D será maior do que 0, mas o valor de DRE pode ainda se aproximar de 1 à medida em que E aumenta para um dado valor de D . De fato, na medida em que E aumenta, é provável que o valor final de D diminua (os erros são filtrados antes de se tornarem defeitos). Se for usada como uma métrica que fornece um indicador da habilidade de filtragem das atividades de controle de qualidade e segurança, a DRE estimula a equipe de projeto de software a instituir técnicas para encontrar o maior número possível de erros antes da entrega do software.

A DRE também pode ser usada no projeto para avaliar a habilidade de uma equipe para encontrar erros antes que eles passem para a próxima atividade na estrutura do software ou para a próxima ação da engenharia de software. Por exemplo, a análise de requisitos produz um modelo de requisitos que pode ser examinado para encontrar e corrigir erros. Aqueles erros que não são detectados durante a revisão do modelo de requisitos passam adiante para o projeto (onde eles podem ser ou não encontrados). Quando usada nesse contexto, redefinimos a DRE como

$$\text{DRE}_i = \frac{E_i}{E_i + E_{i+1}}$$



AVISO
Se a DRE for baixa quando você faz a análise e projeto, dedique algum tempo melhorando a maneira como você executa as revisões técnicas.

onde E_i é o número de erros encontrados durante a ação de engenharia de software i e E_{i+1} é o número de erros encontrados durante a ação de engenharia de software $i + 1$ que estão ligados a erros que não foram descobertos na ação de engenharia de software i .

Um objetivo de qualidade para uma equipe de software (ou um engenheiro de software individual) é conseguir que uma DRE_i que se aproxime de 1. Isto é, os erros deverão ser filtrados antes que eles passem para a próxima atividade ou ação.

CASA SEGURA



Estabelecendo uma abordagem de métricas

A cena: Escritório de Doug Miller, dois dias após a reunião inicial sobre métricas de software.

Os atores: Doug Miller (gerente da equipe de engenharia de software CasaSegura), Vinod Raman e Jamie Lazar, membros da equipe de engenharia de artefatos de software.

Conversa:

Doug: Vocês dois tiveram oportunidade de aprender um pouco sobre métricas de processo e projeto?

Vinod and Jamie: [Ambos acenam a cabeça afirmativamente]

Doug: É sempre uma boa ideia estabelecer metas quando você adota qualquer métrica. Quais são as suas?

Vinod: Nossas métricas deverão focalizar a qualidade. Na verdade, nossa meta geral é manter em um valor mínimo absoluto o número de erros que passamos de uma atividade de engenharia de software para a próxima.

Doug: E ter certeza absoluta de manter o número de defeitos do produto liberado tão próximo de zero quanto possível.

Vinod (acenando afirmativamente): Naturalmente.

Jamie: Eu gosto da DRE como métrica e acho que podemos usá-la para o projeto inteiro, mas também quando passamos de uma atividade estrutural para a próxima. Ela nos estimulará a encontrar erros em cada etapa.

Vinod: Eu gostaria também de coletar o número de horas que gastamos em revisões.

Jamie: E o esforço total gasto em cada tarefa de engenharia de software.

Doug: Você pode calcular uma relação entre a revisão e o desenvolvimento... Pode ser interessante.

Jamie: Eu gostaria também de rastrear alguns dados de casos de uso. Como, por exemplo, o esforço necessário para desenvolver um caso de uso, o esforço necessário para criar software para implementar um caso de uso, e...

Doug (sorrindo): Eu pensava que iríamos manter tudo simples.

Vinod: Deveríamos, mas quando se entra nesse negócio de métricas, há muitas coisas interessantes para ver.

Doug: Eu concordo, mas vamos sem correria e vamos manter nosso objetivo. Limitem os dados a ser coletados em cinco ou seis itens, e estaremos prontos para começar.

25.4 INTEGRANDO MÉTRICAS DENTRO DO PROCESSO DE SOFTWARE

A maioria dos desenvolvedores de software ainda não mede nada, e, infelizmente, muitos têm pouca vontade de começar. Conforme afirmamos anteriormente neste capítulo, o problema é cultural. Tentar coletar medidas onde nenhuma medida foi coletada no passado muitas vezes causa uma resistência. “Por que precisamos fazer isso?”, pergunta o gerente de projeto apressado. “Não vejo razão para isso”, concorda um programador muito atarefado.

Nesta seção, vamos considerar alguns argumentos para métricas de software e apresentar uma abordagem para instituir um programa de coleta de métricas em uma organização de engenharia de software. Mas antes de começar, algumas palavras de prudência (não mais de vinte anos atrás) são sugeridas por Grady e Caswell [Gra87]:

Algumas das coisas que descrevemos aqui parecerão muito fáceis. Na realidade, no entanto, estabelecer um programa bem-sucedido de métricas de software para a empresa inteira é um trabalho difícil. Quando dizemos que você deve esperar pelo menos três anos para conhecer as tendências organizacionais, você tem uma ideia da grandeza desse esforço.

Uma recomendação de cautela sugerida pelos autores é sempre bem-vinda, mas os benefícios das medidas são tão atraentes que compensa o trabalho duro.

25.4.1 Argumentos favoráveis a métricas de software

Por que é tão importante medir o processo de engenharia de software e o artefato que ele produz? A resposta é relativamente óbvia. Se você não medir, não haverá uma maneira real de determinar se está melhorando. E se você não estiver melhorando, está perdido.

Solicitando e avaliando medidas de produtividade e qualidade, uma equipe de software (e seu gerente) pode estabelecer objetivos claros para melhorias do processo de software. Anteriormente neste livro, já destacamos que o software é um assunto estratégico do negócio para muitas empresas. Se o processo através do qual ele é desenvolvido pode ser melhorado, disso pode resultar um impacto direto sobre a base. Mas para estabelecer metas de melhoria, deve ser entendido o status atual do desenvolvimento de software. Consequentemente, são usadas medidas para estabelecer uma linha de base do processo a partir da qual podem ser avaliadas as melhorias.

O rigor do dia a dia do projeto de software deixa pouco tempo para pensamento estratégico. Os gerentes de projeto de software estão preocupados com problemas mais corriqueiros (mas igualmente importantes): desenvolver estimativas claras de projeto, produzir sistemas de qualidade mais alta, entrega do produto dentro do prazo. Usando as medidas para estabelecer uma linha de base de projeto, cada um desses aspectos se torna mais controlável. Já mencionamos que a linha de base serve como um referencial para estimativas. Além disso, a coleção de métricas de qualidade permite que a organização “sintonize” seu processo de software para remover as “poucas causas vitais” de defeitos que têm o maior impacto sobre o desenvolvimento do software.⁹

“Nós controlamos as coisas por números em muitos aspectos da nossa vida... Esses números nos dão uma visão e ajudam a direcionar nossas ações.”

Michael Mah e
Larry Putnam

25.4.2 Estabelecendo uma linha de base

Estabelecendo-se uma linha de base de métricas, podem ser obtidos benefícios nos níveis (técnicos) de processo, projeto e produto. No entanto, as informações que são coletadas não precisam ser fundamentalmente diferentes. As mesmas métricas podem servir a muitos mestres. As linhas de base das métricas consistem em dados coletados de projetos de desenvolvimento de software do passado e pode ser tão simples quanto a tabela apresentada na Figura 25.2 ou tão complexas quanto uma abrangente base de dados contendo dezenas de medidas de projeto e métricas delas derivadas.

Para ser um auxílio eficaz na melhoria do processo e/ou estimativas de custo e esforços, os dados da linha de base devem ter os seguintes atributos: (1) os dados devem ser razoavelmente precisos – “chutes” sobre projetos passados devem ser evitados, (2) devem ser coletados dados de tantos projetos quantos for possível, (3) as medidas devem ser consistentes (Por exemplo, uma linha de código deve ser interpretada consistentemente através de todos os projetos dos quais são coletados os dados), (4) as aplicações devem ser similares ao trabalho que deve ser estimado – faz pouco sentido usar uma linha de base para trabalho com sistemas de informação em lote para estimar uma aplicação em tempo real.

25.4.3 Coleta, cálculo e avaliação de métricas

O processo para estabelecer uma linha de base de métricas está ilustrado na Figura 25.3. Em uma situação ideal, os dados necessários para estabelecer uma referência seriam coletados dinamicamente. Infelizmente, isso raramente acontece. Portanto, a coleta de dados requer uma investigação histórica de projetos passados para reconstruir os dados necessários. Uma vez coletadas as medidas (sem dúvida é a parte mais difícil), é possível o cálculo das métricas. Dependendo da amplitude das medidas coletadas, as métricas podem se estender sobre um amplo intervalo de métricas orientadas a aplicação (por exemplo, LOC, FP, orientada a objeto, WebApp), bem como outras métricas orientadas à qualidade e ao projeto. Finalmente, as métricas devem ser avaliadas e aplicadas durante a estimativa, trabalho técnico, controle de projeto e melhoria de processo. A avaliação de métricas focaliza as razões subjacentes para os resultados obtidos e produz um conjunto de indicadores que guiam o projeto ou o processo.

? O que é uma linha de base de métricas e quais os benefícios que ela proporciona a um engenheiro de software?

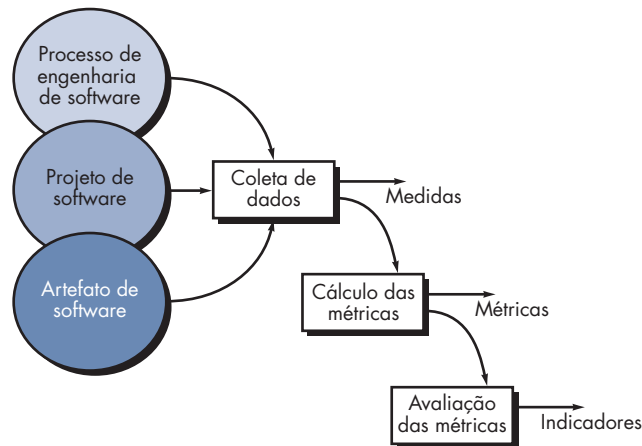
PONTO-CHAVE

Dados das linhas de base de métricas devem ser coletados de uma grande amostragem representativa de projetos de software do passado.

⁹ Essas ideias foram formalizadas em uma abordagem chamada garantia estatística de qualidade de software (*statistical software quality assurance*).

FIGURA 25.3

Processo de coleta de métricas de software



25.5 MÉTRICAS PARA PEQUENAS ORGANIZAÇÕES



AVISO
Se você estiver apenas começando a coletar dados de métricas, lembre-se de mantê-las simples. Se você começar a se afundar nos dados, o seu trabalho com métricas não terá sucesso.

A grande maioria das organizações de desenvolvimento de software têm menos de 20 profissionais de software. Não é razoável, e na maioria dos casos não é viável, esperar que essas organizações desenvolvam programas abrangentes de métricas de software. No entanto, é razoável sugerir que organizações de software de todos os tamanhos meçam e depois usem as métricas resultantes para ajudar a melhorar seu processo local de software e a qualidade e prazo dos produtos que elas produzem.

Um abordagem de bom senso para a implementação de qualquer atividade relacionada com processo de software é: mantenha simples, ajuste para satisfazer as necessidades locais, e certifique-se de que tudo agregue valor. Nos próximos parágrafos, examinaremos como essas diretrizes se relacionam com as métricas para pequenas empresas.¹⁰

“Manter simples” é uma diretriz que funciona razoavelmente bem em muitas atividades. Mas como você poderá derivar um conjunto de métricas de software “simples” que ainda assim tenha valor, e como você pode ter certeza de que essas métricas simples atenderão às necessidades específicas da sua organização de software? Você pode começar focalizando não na medição, mas sim os resultados. O grupo de software é consultado para definir um objetivo único que requer melhorias. Por exemplo, “reduzir o tempo necessário para avaliar e implementar solicitações de alterações”. Uma pequena organização pode escolher o seguinte conjunto de medidas que podem ser obtidas facilmente:

- Tempo (horas ou dias) decorridos desde o instante em que é feita uma solicitação até que a avaliação esteja completa, t_{fila} .
- Esforço (homem-hora) para executar a avaliação, $W_{\text{avaliação}}$.
- Tempo (horas ou dias) decorridos desde o término da avaliação até a atribuição da ordem de alteração para o pessoal, $t_{\text{avaliação}}$.
- Esforço (homem-hora) necessário para fazer a alteração, $W_{\text{alteração}}$.
- Tempo (horas ou dias) para fazer a alteração, $t_{\text{alteração}}$.
- Erros descobertos durante o trabalho para fazer a alteração, $E_{\text{alteração}}$.
- Defeitos descobertos depois que a alteração é liberada para o cliente, $D_{\text{alteração}}$.

Uma vez coletadas essas medidas para um conjunto de solicitações de alteração, é possível calcular o tempo total decorrido desde a solicitação de alteração até a implementação da alteração e a porcentagem de tempo decorrido gasto na classificação inicial, avaliação e atribuição

¹⁰ Essa discussão é igualmente relevante para equipes de software que tenham adotado um processo ágil de desenvolvimento de software (Capítulo 3).

Como devemos derivar um conjunto de métricas de software “simples”?

da mudança, e implementação da alteração. De forma semelhante, pode ser determinada a porcentagem do trabalho necessário para a avaliação e implementação. Essas métricas podem ser analisadas no contexto de dados de qualidade, $E_{\text{alteração}}$ e $D_{\text{alteração}}$. A porcentagem permite visualizar onde o processo de solicitação de alteração se retarda e permite conduzir às etapas de melhoria de processo para reduzir t_{fila} , $W_{\text{avaliação}}$, $t_{\text{avaliação}}$, $W_{\text{alteração}}$, e/ou $E_{\text{alteração}}$. Além disso, a eficiência na remoção de defeitos pode ser computada como

$$\text{DRE} = \frac{E_{\text{alteração}}}{E_{\text{alteração}} + D_{\text{alteração}}}$$

A DRE pode ser comparada com o tempo decorrido e o trabalho total para determinar o impacto das atividades de garantia de qualidade sobre o tempo e trabalho necessários para fazer uma alteração.

Para pequenos grupos, o custo da coleta de medidas e cálculo das métricas varia de 3 a 8 por cento do orçamento do projeto durante a fase de aprendizado e depois cai para menos de 1 por cento do orçamento do projeto depois que os engenheiros de software e gerentes de projeto se familiarizaram com o programa de métricas [Gra99]. Esses custos podem apresentar um substancial retorno sobre o investimento se as informações derivadas dos dados das métricas levarem a melhorias significativas do processo para a organização de software.

25.6 ESTABELECENDO UM PROGRAMA DE MÉTRICAS DE SOFTWARE

O Software Engineering Institute desenvolveu um guia abrangente [Par96b] para estabelecer um programa de métricas de software “orientado a metas”. O livro sugere as seguintes etapas:

1. Identifique as suas metas de negócio.
2. Identifique o que você quer saber ou aprender.
3. Identifique as suas submetas.
4. Identifique as entidades e atributos relacionados com as suas submetas.
5. Formalize as suas metas de medição.
6. Identifique questões quantificáveis e os indicadores relacionados que você usará para ajudá-lo a atingir as suas metas de medição.
7. Identifique os elementos de dados que você coletará para construir os indicadores que ajudam a responder às suas questões.
8. Defina as medidas a ser usadas e torne essas definições operacionais.
9. Identifique as ações que você tomará para implementar as medidas.
10. Prepare um plano para implementar as medidas.

Uma discussão detalhada dessas etapas se encontra no guidebook da SEI. No entanto, temos aqui uma visualização rápida dos pontos principais.

Devido ao fato de que o software suporta as funções de negócio, diferencia sistemas ou produtos baseados em computadores, ou age como um produto por si mesmo, os objetivos definidos para os negócios podem quase sempre ser ligados a metas específicas em nível de engenharia de software. Por exemplo, considere o produto *CasaSegura*. Trabalhando como uma equipe, a engenharia de software e os gerentes de negócio desenvolvem uma lista de metas comerciais com prioridades:

1. Melhorar a satisfação dos seus clientes com seus produtos.
2. Tornar os seus produtos mais fáceis de usar.
3. Reduzir o tempo necessário para ter um novo produto no mercado.
4. Tornar mais fácil o suporte ao nosso produto.
5. Melhorar nossa lucratividade geral.

WebRef

Um Guidebook for Goal Driven Software Measurement pode ser baixado do endereço www.sei.cmu.edu.

PONTO-CHAVE

As métricas de software que você escolhe deverão ser motivadas pelas metas de negócio e técnicas que você deseja atingir.

A organização de software examina cada meta de negócio e pergunta: “Quais as atividades que nós gerenciamos, executamos, ou suportamos e o que nós queremos melhorar nessas atividades?” Para responder a essas perguntas a SEI recomenda a criação de uma “lista entidade-questão” na qual todas as coisas (entidades) dentro do processo de software que são gerenciadas ou influenciadas pela organização de software são listadas. Exemplos de entidades incluem recursos de desenvolvimento, produtos acabados, código fonte, test cases, solicitações de alteração, tarefas de engenharia de software, e cronogramas. Para cada entidade listada, os profissionais do software desenvolvem uma série de questões que investigam características quantitativas da entidade (por exemplo, tamanho, custo, tempo para desenvolver). As questões originadas em consequência da criação de uma lista entidade-questão levam à criação de uma série de submetas que se relacionam diretamente com as entidades criadas e com as atividades executadas como parte do processo de software.

Considere a quarta meta: “Tornar mais fácil o suporte ao nosso produto.” Para essa meta, pode ser criada a seguinte lista de questões [Par96b]:

- As solicitações de alterações do cliente contém as informações que precisamos para avaliar adequadamente a alteração e então implementá-la dentro do prazo normal?
- Qual é o acúmulo de solicitações de alteração?
- O nosso tempo de resposta para corrigir os bugs é aceitável com base nas necessidades do cliente?
- O nosso processo de controle de alterações (Capítulo 22) é seguido?
- As alterações de alta prioridade são implementadas dentro do prazo normal?

Com base nessas questões, a organização de software pode derivar a seguinte submeta: *Melhorar o desempenho do processo de gerenciamento de alterações.* As entidades e atributos do

INFORMAÇÕES



Estabelecendo um programa de métricas

O Software Productivity Center (www.spc.ca) sugere uma abordagem de oito passos para estabelecer um programa de métricas em uma organização de software que pode ser usada como alternativa para a abordagem SEI descrita na Seção 25.6. A abordagem se resume em:

1. Entender o processo de software existente.
 - São identificadas as atividades da estrutura (Capítulo 2).
 - São descritas as informações para cada atividade.
 - São definidas as tarefas associadas com cada atividade.
 - São descritas as funções de garantia de qualidade.
 - São listados os produtos acabados que são produzidos.
2. Definir as metas a ser atingidas estabelecendo um programa de métricas.
 - Exemplos:
 - melhorar a precisão da estimativa,
 - melhorar a qualidade do produto.
3. Identificar métricas necessárias para atingir as metas.
 - São definidas as questões a ser respondidas; por exemplo, *Quantos erros encontrados em uma atividade estrutural podem estar ligados com a atividade estrutural precedente?*
 - Criar medidas e métricas que ajudarão a responder a essas questões.
4. Identificar as medidas e métricas a ser coletadas e computadas.

5. Estabelecer um processo de coleta de medidas respondendo às seguintes questões:

Qual é a fonte das medições?

Podem ser usadas ferramentas para coletar dados?

Quem é responsável pela coleta de dados?

Quando os dados são coletados e registrados?

Como os dados são armazenados?

Que mecanismos de validação são usados para garantir que os dados estão corretos?

6. Adquirir as ferramentas apropriadas para ajudar na coleta e avaliação.

7. Estabelecer uma base de dados de métricas.

É definida a sofisticação relativa da base de dados.

É explorado o uso de ferramentas relacionadas (por exemplo, um repositório SCM, Capítulo 26).

São avaliados os produtos existentes da base de dados.

8. Definir mecanismos de feedback apropriados.

Quem solicita informações das métricas em andamento?

Para quem as informações devem ser fornecidas?

Qual é o formato das informações?

Uma descrição consideravelmente mais detalhada dessas oito etapas pode ser baixada do site www.spc.ca/resources/metrics/.

processo de software que são relevantes à submeta são identificadas, e são delineadas as metas de medição associadas com elas.

A SEI [Par96b] fornece instruções detalhadas para as etapas 6 a 10 de sua abordagem de medição motivada por meta. Essencialmente, você refina as metas de medição, transformando-as em questões que são mais refinadas tornando-se entidades e atributos que são então refinados tornando-se métricas.

25.7 RESUMO

As medidas permitem aos gerentes e profissionais melhorar o processo de software; ajudam no planejamento, acompanhamento e controle dos projetos de software; e avaliam a qualidade do artefato de software que é produzido. Medidas de atributos específicos de processo, projeto e produto são usadas para computar as métricas de software. Essas métricas podem ser analisadas para fornecer indicadores que guiam a gerência e as ações técnicas.

As métricas de processo permitem que uma organização tenha uma visão estratégica, fornecendo informações sobre a eficácia de um processo de software. Métricas de projeto são táticas. Elas permitem que o gerente de projeto adapte o workflow de projeto e a abordagem técnica de uma maneira em tempo real.

Na indústria, são usadas métricas orientadas a tamanho e função. Métricas orientadas a tamanho usam a quantidade de linhas de código como um fator de normalização para outras medidas como pessoa-mês ou defeitos. O ponto de função é derivado de medidas de domínio de informações e de uma avaliação subjetiva da complexidade do problema. Além disso, podem ser usadas métricas orientadas a objeto e métricas orientadas a aplicação Web.

Métricas de qualidade de software, como as métricas de produtividade, focalizam o processo, o projeto e o produto. Desenvolvendo e analisando uma linha de base de métricas para qualidade, uma organização pode corrigir aquelas áreas do processo de software que são a causa dos defeitos de software.

Medidas resultam em mudanças culturais. Coleta de dados, cálculo das métricas, e análise das métricas são as três etapas que devem ser implementadas para começar um programa de métricas. Em geral, uma abordagem motivada por meta ajuda uma organização a focalizar as métricas corretas para seus negócios. Criando uma linha de base de métricas – uma base de dados contendo medidas de processo e produto – os engenheiros de software e seus gerentes podem ter uma visão melhor do trabalho que eles executam e do produto que eles produzem.

PROBLEMAS E PONTOS A PONDERAR

- 25.1.** Descreva a diferença entre métricas de processo e de projeto com suas próprias palavras.
- 25.2.** Por que algumas métricas devem ser mantidas “privadas”? Forneça exemplos de três métricas que deverão ser privadas. Forneça exemplos de três métricas que deverão ser públicas.
- 25.3.** O que é uma medida indireta, e por que essas medidas são comuns no trabalho com métricas de software?
- 25.4.** Grady sugere uma etiqueta para métricas de software. Você pode acrescentar mais três regras àquelas mencionadas na Seção 25.1.1?
- 25.5.** A Equipe A encontrou 342 erros durante um processo de engenharia de software antes do lançamento. A Equipe B encontrou 184 erros. Que medidas adicionais terão de ser feitas para os projetos A e B para determinar qual das equipes eliminou erros com mais eficiência? Que métricas você poderia propor para ajudar nessa determinação? Que dados históricos podem ser úteis?
- 25.6.** Apresente um argumento contra a adoção de número de linhas de código como medida de produtividade de software. O seu argumento poderá se manter quando forem consideradas dezenas ou centenas de projetos?

25.7. Calcule o valor de pontos de função para um projeto com as seguintes características no domínio de informações:

Número de entradas de usuário: 32

Número de saídas de usuário: 60

Número de consultas de usuário: 24

Número de arquivos: 8

Número interfaces externas: 2

Suponha que todos os valores de ajuste de complexidade são médios. Use o algoritmo do Capítulo 23.

25.8. Usando a tabela apresentada na Seção 25.2.3, apresente um argumento contra o uso de linguagem assembler com base na funcionalidade fornecida pelas instruções de código. Novamente referindo-se à tabela, discuta porque C++ seria uma alternativa melhor do que C.

25.9. O software usado para controlar uma fotocopiadora requer 32.000 linhas de C e 4.200 linhas de Smalltalk. Estime o número de pontos de função do software que está na copiadora.

25.10. Uma equipe de engenharia Web criou um WebApp e-commerce contendo 145 páginas individuais. Dessas páginas, 65 são dinâmicas, ou seja, elas são geradas internamente com base em entradas do usuário. Qual é o índice de personalização dessa aplicação?

25.11. Um WebApp e seu ambiente de suporte não foram totalmente protegidos contra ataques. Os engenheiros Web estimam que a probabilidade de repelir um ataque é de apenas 30%. O sistema não contém informações sensíveis ou comprometedoras, portanto, a probabilidade de ataque é 25%. Qual é a integridade do WebApp?

25.12. Na conclusão de um projeto, foi determinado que foram encontrados 30 erros durante a atividade de modelagem e foram encontrados 12 erros durante a atividade de construção que estavam ligados a erros que não foram descobertos na atividade de modelagem. Qual é a DRE para a atividade de modelagem?

25.13. Uma equipe de software fornece um incremento de software para usuários finais. Os usuários descobrem 8 defeitos durante o primeiro mês de uso. Antes da entrega, a equipe de software encontrou 242 erros durante as revisões técnicas formais e em todas as tarefas de teste. Qual é a DRE geral do projeto após um mês de uso?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A melhoria do processo de software (software process improvement – SPI) tem recebido uma atenção significativa durante as últimas duas décadas. Como as medições e as métricas de software são fundamentais para uma melhora bem-sucedida do processo de software, muitos livros sobre SPI também discutem métricas. Rico (*ROI of Software Process Improvement*, J. Ross Publishing, 2004) fornece uma discussão aprofundada da SPI e as métricas que podem ajudar uma organização a atingi-la. Ebert e seus colegas (*Best Practices in Software Measurement*, Springer, 2004) trata do uso das medições dentro do contexto das normas ISO e CMMI. Kan (*Metrics and Models in Software Quality Engineering*, 2d ed., Addison-Wesley, 2002) apresenta uma coleção de métricas relevantes.

Ebert e Dumke (*Software Measurement*, Springer, 2007) proporcionam um bom tratamento sobre medidas e métricas de como elas devem ser aplicadas para projetos TI. McGarry e seus colegas (*Practical Software Measurement*, Addison-Wesley, 2001) apresenta uma profunda consultoria para avaliação de processo de software. Uma boa coleção de artigos foi editada por Haug e seus colegas (*Software Process Improvement: Metrics, Measurement, and Process Modeling*, Springer-Verlag, 2001). Florac e Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) e Fenton e Pfleeger (*Software Metrics: A Rigorous and Practical Approach*, Revised, Brooks/Cole Publishers, 1998) discutem como as métricas de software podem ser usadas para proporcionar os indicadores necessários para melhorar o processo de software.

Laird e Brennan (*Software Measurement and Estimation*, Wiley-IEEE Computer Society Press, 2006) e Goodman (*Software Metrics: Best Practices for Successful IT Management*, Rothstein Associates, Inc., 2004) discutem o uso das métricas de software para gerenciamento de projeto e estimativas. Putnam e Myers (*Five Core Metrics*, Dorset House, 2003) usam uma base de dados de mais de 6.000 projetos de software para demonstrar como cinco métricas fundamentais – tempo, trabalho, tamanho, confiabilidade e produtividade de processo – podem ser usadas para controlar projetos de software. Maxwell (*Applied Statistics for Software Managers*, Prentice-Hall, 2003) apresenta técnicas para analisar dados de projeto de software. Munson (*Software Engineering Measurement*, Auerbach, 2003) discute um amplo conjunto de aspectos de medidas em engenharia de software. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) descreve fatores quantitativos e qualitativos que ajudam uma organização a avaliar seu processo e práticas de software.

A medida de pontos de função tornou-se uma técnica amplamente utilizada em muitas áreas da engenharia de software. Parthasarathy (*Practical Software Estimation: Function Point Methods for Insourced and Outsourced Projects*, Addison-Wesley, 2007) fornece um guia abrangente. Garmus e Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) discutem métricas de processo com ênfase na análise de pontos de função.

Relativamente pouco se tem publicado sobre métricas em trabalhos de engenharia Web. Kaushik (*Web Analytics: An Hour a Day*, Sybex, 2007), Stern (*Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002), Inan e Kean (*Measuring the Success of Your Website*, Longman, 2002), e Nobles e Grady (*Web Site Analysis and Reporting*, Premier Press, 2001) tratam das métricas Web a partir de uma perspectiva de negócios e marketing.

A mais recente pesquisa na área de métricas está resumida pelo IEEE (*Symposium on Software Metrics*, publicado anualmente). Uma grande variedade de fontes de informações sobre métricas de processo e projeto está disponível na Internet. Uma lista atualizada das referências da World Wide Web relevantes a métricas de processo e projeto pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

**CONCEITOS-
-CHAVE**

escopo do	
software	606
equação do software	621
estimativa	605
ágil	622
baseada em FP	612
baseadas em	
problemas	611
baseadas em	
processo	614
casos de uso	616
modelos	
empíricos	618
projetos orientados	
a objeto	622
reconciliação	617
WebApps	623
planejamento do	
projeto	606
viabilidade	606

O gerenciamento de projeto de software começa com uma série de atividades chamadas coletivamente de *planejamento de projeto*. Antes de iniciar o projeto, a equipe de software deverá fazer uma estimativa do trabalho, os recursos que serão necessários e o tempo necessário para a sua conclusão. Uma vez executadas essas atividades, a equipe de projeto deverá estabelecer um cronograma que defina as tarefas de engenharia de software e as metas, deverá identificar os responsáveis pela execução de cada tarefa e especificar as dependências entre tarefas que podem ter forte influência no progresso do trabalho.

Em um excelente guia de “sobrevivência de projeto de software”, Steve McConnell [McC98] apresenta uma visão do planejamento de projeto na prática:

Muitos profissionais técnicos gostariam de ir direto ao trabalho em vez de passar o tempo planejando. Muitos gerentes técnicos não possuem treinamento suficiente em gerenciamento para se sentirem confiantes de que o seu planejamento melhorará os resultados de um projeto. Como nenhuma das partes quer realizar planejamento, a tarefa em geral acaba não sendo cumprida.

Mas não planejar é uma das falhas mais críticas que um projeto pode ter... O planejamento eficaz é necessário para resolver problemas precoces [no início do projeto] com baixo custo, em vez de problemas tardios [que surgem mais tarde no projeto] com alto custo. Os projetos consomem em média 80% do tempo em retrabalho – corrigindo erros que foram feitos no início do projeto.

PANORAMA

O que é? Uma necessidade real de software foi estabelecida; as pessoas interessadas estão envolvidas, os engenheiros de software estão prontos para começar, e o projeto está para ser iniciado. E como você procede? O planejamento de projeto de software abrange cinco atividades importantes – estimativa, cronograma, análise de risco, planejamento do gerenciamento de qualidade e planejamento para o gerenciamento de alterações. Neste capítulo, consideraremos somente as estimativas – a tentativa para determinar quanto dinheiro, esforço, recursos e tempo serão necessários para criar um sistema ou artefato baseado em software específico.

Quem realiza? Os gerentes de projeto de software – usando informações recebidas dos interessados no projeto e dados das métricas de software coletados de projetos anteriores.

Por que é importante? Você construiria uma casa sem saber quanto está disposto a gastar, as tarefas que precisam ser executadas e os prazos para executar o trabalho? É claro que não, e como a maioria dos sistemas e produtos baseados em computadores custa consideravelmente mais para ser construídos do que a construção de

uma grande casa, parece bem razoável desenvolver uma estimativa antes de começar a criar o software.

Quais são as etapas envolvidas? As estimativas começam com uma descrição do escopo do problema. O problema é então decomposto em uma série de questões menores, e cada um desses problemas é estimado por meio de dados históricos e da experiência como guia. A complexidade e os riscos do problema são considerados antes que a estimativa final seja feita.

Qual é o artefato? É gerada uma tabela simples que descreve as tarefas a ser executadas, as funções a ser implementadas, e o custo, esforço e tempo envolvidos para cada atividade.

Como garantir que o trabalho foi executado corretamente? Isso é difícil, porque não se pode realmente saber até que o projeto seja finalizado. No entanto, se você tiver experiência e seguir uma abordagem sistemática, se gerar estimativas usando dados históricos confiáveis, se criar dados de estimativa usando pelo menos dois métodos diferentes, se estabelecer um cronograma realístico e se adaptá-lo continuamente à medida que o projeto avança, pode-se ter certeza de que foi feita a melhor escolha.

McConnell alega que toda equipe pode arranjar espaço para planejar (e adaptar o plano durante o projeto) simplesmente reservando uma pequena porcentagem do tempo que teria sido gasto no retrabalho porque o planejamento não foi feito.

26.1 OBSERVAÇÕES E ESTIMATIVAS

O planejamento requer que você assuma um comprometimento inicial, mesmo que este venha a demonstrar mais tarde que estava errado. Sempre que forem feitas estimativas, deve-se olhar o futuro e aceitar certo grau de incerteza. Segundo Frederick Brooks [Bro95]:

...Nossas técnicas de estimativa são muito mal desenvolvidas. E, pior, refletem uma suposição bastante falsa, não declarada, de que tudo sairá muito bem... Porque não temos certeza sobre nossas estimativas, gerentes de software muitas vezes não possuem a firmeza necessária para fazer as pessoas esperarem por um bom produto.

Embora estimar seja muito mais arte do que ciência, não precisa ser conduzida de maneira aleatória. Existem técnicas úteis para estimar tempo e esforço. As métricas de projeto e processo podem proporcionar perspectivas históricas e valiosas informações para gerar estimativas quantitativas. A experiência (de todos os envolvidos) pode ajudar imensamente à medida que as estimativas são desenvolvidas e revisadas. Por serem a base para todas as outras ações do planejamento de projeto, e pelo fato de o planejamento de projeto fornecer a direção para uma engenharia de software bem-sucedida, seria uma péssima ideia iniciar sem as estimativas.

As estimativas de recursos, custos e cronograma para um trabalho de engenharia de software requerem experiência, acesso a boas informações históricas (métricas), e a coragem de se comprometer com as previsões quantitativas quando tudo o que existe são apenas informações qualitativas. A estimativa traz um risco inerente,¹ e esse risco leva à incerteza.

A *complexidade do projeto* tem um forte efeito sobre a incerteza inerente ao planejamento. No entanto, é uma medida relativa afetada pela familiaridade com esforços passados. Alguém que pela primeira vez desenvolve uma sofisticada aplicação para comércio eletrônico pode considerá-la excessivamente complexa. No entanto, uma equipe de engenharia para Web desenvolvendo sua décima aplicação WebApp para comércio eletrônico consideraria isso um trabalho comum. Já foram propostas várias medidas de complexidade quantitativa de software [Zus97]. Estas são aplicadas em nível de projeto ou código e são, portanto, difíceis de usar durante o planejamento de software (antes do projeto e código). No entanto, outras avaliações mais subjetivas de complexidade (por exemplo, fatores de ajuste de complexidade de pontos de função descritos no Capítulo 23) podem ser estabelecidas no início do processo de planejamento.

O *tamanho do projeto* é outro fator importante que pode afetar a precisão e a eficácia das estimativas. À medida que o tamanho aumenta, a interdependência entre os vários elementos do software cresce rapidamente.² A decomposição do problema, uma abordagem importante para a estimativa, torna-se mais difícil, porque o refinamento dos elementos envolvidos pode ainda ser considerável. Parafraseando a lei de Murphy: “O que pode sair errado, sairá errado” – e se houver mais itens que podem falhar, mais itens falharão.

O *grau de incerteza estrutural* também tem um efeito sobre o risco das estimativas. Nesse contexto, a estrutura refere-se ao grau segundo o qual os requisitos foram solidificados, a facilidade com a qual as funções podem ser separadas e a natureza hierárquica das informações a ser processadas.

A disponibilidade de informações históricas tem uma forte influência sobre o risco das estimativas. Utilizar procedimentos que funcionaram pode melhorar áreas problemáticas. Quando

“Boas abordagens de estimativa e dados históricos sólidos são a maior esperança de que a realidade vencerá as demandas impossíveis.”

Caper Jones

PONTO-CHAVE

A complexidade do projeto, tamanho e grau de incerteza estrutural afetam a confiabilidade das estimativas.

¹ Técnicas sistemáticas para análise de riscos são apresentadas no Capítulo 28.

² O tamanho muitas vezes aumenta devido ao “deslizamento do escopo” que ocorre quando os requisitos do problema mudam. O aumento no tamanho do projeto pode ter um impacto geométrico sobre o custo e o cronograma do projeto (Michael Mah, comunicação pessoal).

“É característica de uma mente instruída satisfazer-se com o grau de precisão que a natureza do assunto permite, e não procurar exatidão quando apenas uma aproximação da verdade é possível.”

Aristóteles

há disponíveis métricas de software abrangentes (Capítulo 25) para projetos passados, as estimativas podem ser feitas com maior segurança, podem ser estabelecidos os cronogramas para evitar dificuldades passadas, e o risco em geral é reduzido.

O risco das estimativas é medido pelo grau de incerteza nas estimativas quantitativas estabelecidas para os recursos, custo e cronograma. Se o escopo do projeto é mal entendido ou se os requisitos do projeto sofrem alterações, a incerteza e o risco das estimativas tornam-se perigosamente altos. Como planejador, você e o cliente devem reconhecer que variabilidade nos requisitos de software significa instabilidade nos custos e no cronograma.

No entanto, você não deve se tornar obsessivo em relação às estimativas. Abordagens modernas de engenharia de software (por exemplo, modelos incrementais de processo) assumem uma visão iterativa do desenvolvimento. Em tais abordagens, é possível – embora nem sempre politicamente aceitável – voltar à estimativa (conforme mais informações são conhecidas) e revisá-la quando o cliente fizer alterações nos requisitos.

26.2 O PROCESSO DE PLANEJAMENTO DO PROJETO



Quanto mais você sabe, melhor você estima. Portanto, atualize suas estimativas à medida que o projeto avança.

O objetivo do planejamento de software é proporcionar uma estrutura que permite ao gerente fazer estimativas razoáveis dos recursos, custo e cronograma. Além disso, as estimativas devem tentar definir cenários de melhor e pior caso para que os resultados do projeto possam ser delineados. Embora haja um grau inerente de incerteza, a equipe de software participa de um plano estabelecido como consequência dessas tarefas. O plano deve ser adaptado e atualizado à medida que o projeto avança. Nas próximas seções, discutiremos cada uma das ações associadas a planejamento de projeto de software.

CONJUNTO DE TAREFAS



Conjunto de tarefas para planejamento de projeto

1. Estabeleça o escopo do projeto.
2. Determine a viabilidade.
3. Analise os riscos (Capítulo 28).
4. Defina os recursos necessários.
 - a. Determine os recursos humanos necessários.
 - b. Defina recursos de software reutilizáveis.
 - c. Identifique recursos ambientais.
5. Estime o custo e a mão de obra.
 - a. Decomponha o problema.
 - b. Desenvolva duas ou mais estimativas usando tamanho, pontos de função, tarefas de processo ou casos de uso.
 - c. Reconcilie as estimativas.
6. Desenvolva um cronograma de projeto (Capítulo 27).
 - a. Estabeleça um conjunto significativo de tarefas.
 - b. Defina uma rede de tarefas.
 - c. Use ferramentas de cronograma para desenvolver um diagrama de tempos.
 - d. Defina mecanismos para acompanhamento do cronograma.

26.3 ESCOPO E VIABILIDADE DO SOFTWARE

O *escopo do software* descreve as funções e características que devem ser fornecidas aos usuários finais; os dados que entram e saem; o “conteúdo” que é apresentado aos usuários como consequência do uso do software; e o desempenho, restrições, interfaces e confiabilidade que *limitam* o sistema. O escopo é definido por meio de uma das duas técnicas:

1. Uma descrição narrativa do escopo do software é desenvolvida após comunicação com todos os interessados.
2. É desenvolvida uma série³ de casos de uso pelos usuários finais.

3 Casos de uso foram discutidos em detalhe na Parte 2 deste livro. Um caso de uso é uma descrição baseada em cenário da interação do usuário com o software sob o ponto de vista do usuário.



A viabilidade do projeto é importante, mas uma consideração das necessidades do negócio é ainda mais importante. Não é uma boa ideia criar um sistema ou produto de alta tecnologia que ninguém quer.

Funções descritas na definição do escopo (ou nos casos de uso) são avaliadas e em algumas situações refinadas para fornecer mais detalhes antes de iniciar as estimativas. Como as estimativas de custo e cronograma são ambas funcionalmente orientadas, muitas vezes é aconselhável certo grau de decomposição. As considerações de desempenho abrangem requisitos de processamento e tempo de resposta. As restrições identificam limites colocados no software por hardware externo, memória disponível ou outros sistemas existentes.

Uma vez identificado o escopo (com a participação do cliente), é razoável perguntar: “Podemos criar software que atenda a esse escopo? O projeto é viável?”. Muitas vezes, os engenheiros de software passam rapidamente por essas questões (ou são empurrados por gerentes impacientes ou outras pessoas interessadas), somente para se envolverem em um projeto que já está condenado desde o início. Putnam e Myers [Put97a] tratam do assunto quando escrevem:

Nem tudo o que é imaginável é viável, nem mesmo no software, transiente como pode parecer para quem está de fora. Por outro lado, a viabilidade do software tem quatro dimensões sólidas: *Tecnologia* – O projeto é tecnicamente viável? Ele está incluído no estado da arte? Os erros podem ser reduzidos até um nível que corresponda às necessidades da aplicação? *Finanças* – Ele é financeiramente viável? O desenvolvimento pode ser completado a um custo que a organização de software, seus clientes ou o mercado podem pagar? *Tempo* – O tempo que o projeto leva para atingir o mercado vence a concorrência? *Recursos* – A organização tem os recursos necessários para ser bem-sucedida?

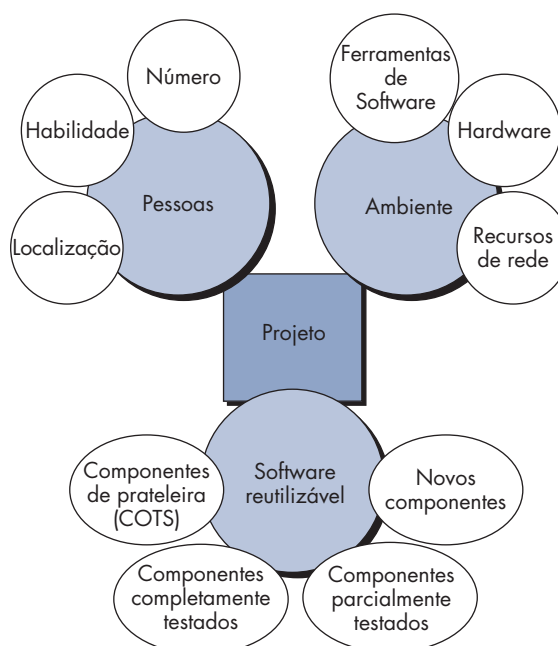
Putnam e Myers sugerem corretamente que só escopo não é suficiente. Uma vez entendido o escopo, deve-se trabalhar para determinar se ele pode ser feito segundo as dimensões que acabamos de descrever. Essa é uma parte crucial do processo de estimativas, que muitas vezes passa despercebida.

26.4 RECURSOS

A segunda tarefa do planejamento é a estimativa dos recursos necessários para executar o trabalho de desenvolvimento de software. A Figura 26.1 mostra as três principais categorias de recursos de engenharia de software – pessoas, componentes de software reutilizáveis e o ambiente de desenvolvimento (hardware e ferramentas de software). Cada recurso é especificado

FIGURA 26.1

Recursos de projeto



com quatro características: descrição do recurso, uma definição da disponibilidade, instante em que o recurso será necessário e tempo durante o qual o recurso será requerido. As duas últimas podem ser vistas como uma *janela de tempo*. A disponibilidade do recurso para uma janela de tempo especificada deve ser estabelecida o mais cedo possível.

26.4.1 Recursos humanos

O planejador começa avaliando o escopo do software e selecionando as habilidades necessárias para completar o desenvolvimento. São especificadas a posição organizacional (por exemplo, gerente, engenheiro de software sênior) e especialização (por exemplo, telecomunicações, base de dados, cliente-servidor). Para projetos relativamente pequenos (poucas pessoas-mês), um único profissional pode executar todas as tarefas de engenharia de software, consultando os especialistas quando necessário. Para projetos maiores, a equipe pode estar geograficamente dispersa em localizações diferentes. Portanto, é especificada a localização de cada recurso humano.

O número de pessoas necessárias para um projeto de software pode ser determinado somente após uma estimativa do esforço de desenvolvimento (por exemplo, pessoas-mês). As técnicas para estimar o esforço são discutidas mais adiante neste capítulo.

26.4.2 Recursos de software reutilizáveis

A engenharia de software baseada em componente (*component-based software engineering* – CBSE)⁴ enfatiza a reusabilidade – isto é, a criação e reutilização de blocos básicos de software. Esses blocos básicos, chamados também de *componentes*, devem ser catalogados para facilitar a referência, padronizados para facilitar a aplicação e validados para facilitar a integração. Benatan [Ben00] sugere quatro categorias de recursos de software que deverão ser consideradas conforme progride o planejamento:



Nunca se esqueça de que integrar uma variedade de componentes reutilizáveis pode ser um desafio significativo. Pior ainda, o problema da integração reaparece à medida que vários componentes são atualizados.

Componentes de prateleira. Software existente que pode ser adquirido de terceiros ou de projetos anteriores. Componentes de prateleira (*commercial off-the-shelf* – COTS) são comprados de terceiros, prontos para uso no projeto atual e que foram totalmente validados.

Componentes totalmente testados. Especificações, projetos, código ou dados de teste desenvolvidos para projetos anteriores, similares ao software a ser criado para o projeto atual. Os membros da equipe de software atual têm experiência plena na área de aplicação representada por esses componentes. Portanto, as modificações necessárias em componentes totalmente testados apresentarão um risco relativamente baixo.

Componentes parcialmente testados. Especificações, projetos, código ou dados de teste desenvolvidos para projetos anteriores que estão relacionados com o software a ser criado para o projeto atual, mas que necessitará de uma grande modificação. Os membros da equipe de software atual têm uma experiência limitada na área de aplicações representada por esses componentes. Portanto, as modificações necessárias para componentes parcialmente testados terão um grau razoável de risco.

Novos componentes. Componentes de software que devem ser criados pela equipe especificamente para as necessidades do projeto atual.

Ironicamente, os componentes de software reutilizáveis são muitas vezes negligenciados durante o planejamento, só para se tornarem uma preocupação suprema mais adiante no processo de software. É aconselhável especificar requisitos de recursos de software o quanto antes. Dessa maneira, pode ser feita a avaliação técnica das alternativas e a aquisição realizada a tempo.

26.4.3 Recursos de ambiente

O ambiente que suporta um projeto de software, muitas vezes chamado de *ambiente de engenharia de software* (*software engineering environment* – SEE), incorpora hardware e soft-

⁴ CBSE é tratado no Capítulo 10.

ware. O hardware fornece uma plataforma que suporta as ferramentas (software) necessárias para produzir os artefatos resultantes de uma boa prática de engenharia de software.⁵ Devido a muitas organizações de software terem múltiplas localizações que requerem acesso ao SEE, deve-se prescrever a janela de tempo necessária para hardware e software e verificar se esses recursos estarão disponíveis.

Quando um sistema baseado em computador (incorporando hardware e software especializado) deve ser desenvolvido, a equipe pode necessitar de acesso a elementos de hardware que estão sendo desenvolvidos por outras equipes de engenharia. Por exemplo, o software para um dispositivo robótico usado em uma célula de manufatura pode necessitar de um robô específico (por exemplo, um soldador robótico) como parte da etapa de teste de validação; um projeto avançado para layout de página pode necessitar de um sistema digital de impressão de alta velocidade em algum instante durante o desenvolvimento. Cada elemento de hardware precisa ser especificado como parte do planejamento.

26.5 ESTIMATIVA DO PROJETO DE SOFTWARE

“Em uma época de terceirização e concorrência cada vez maior, a habilidade para estimar com mais precisão... emergiu como um fator crítico de sucesso para muitos grupos de TI.”

Rob Thomsett

As estimativas de custo e esforço de software nunca serão uma ciência exata. Muitas variáveis – fatores humanos, técnicos, ambientais e políticos – podem afetar o custo final do software e o esforço necessário para desenvolvê-lo. No entanto, as estimativas de projeto de software podem ser transformadas de algo sobrenatural para uma série de etapas sistemáticas que proporcionam estimativas com um risco aceitável. Para conseguir estimativas confiáveis de custo e esforço, surge uma série de opções:

1. Adie a estimativa no decorrer do projeto (obviamente, podemos conseguir uma estimativa com precisão de 100% depois que o projeto estiver completo!).
2. Fundamente suas estimativas em projetos similares que já foram completados.
3. Use técnicas de decomposição relativamente simples para gerar estimativas de custo de projeto e esforço.
4. Use um ou mais modelos empíricos para estimativa de custo e esforço do software.

Infelizmente, a primeira opção, embora atrativa, não é prática. As estimativas de custo devem ser feitas no início. No entanto, é preciso reconhecer que quanto mais esperar, mais informação você terá da realidade, e menor será a probabilidade de que cometa erros graves nas suas estimativas.

A segunda opção pode funcionar razoavelmente bem, se o projeto atual for muito similar a trabalhos anteriores e outras influências de projeto (por exemplo, o cliente, as condições comerciais, o ambiente de engenharia de software, prazos finais) forem quase equivalentes. Infelizmente, a experiência não tem sido sempre um bom indicador de resultados futuros.

As demais opções são abordagens viáveis para estimativa de projeto de software. Idealmente, as técnicas mencionadas para cada opção deverão ser aplicadas em paralelo; cada uma sendo usada para verificar a outra. As técnicas de decomposição assumem uma abordagem do tipo “dividir para conquistar” para a estimativa do projeto de software. Decompondo um projeto em suas funções principais e atividades de engenharia de software relacionadas, as estimativas de custo e esforço podem ser feitas passo a passo. Podem ser usados modelos empíricos de estimativa para complementar as técnicas de decomposição e oferecer uma abordagem de estimativa potencialmente valiosa por si mesma. Um modelo é baseado na experiência (dados históricos) e toma a seguinte forma:

$$d = f(v_i)$$

“É muito difícil fazer uma defesa vigorosa, plausível e arriscada de uma estimativa não originada de um método quantitativo, suportada por dados escassos e certificada principalmente pelas pressões dos gerentes.”

Fred Brooks

5 Outro hardware – o *ambiente-alvo* – é o computador no qual o software será executado quando liberado para o cliente final.

em que d é um entre uma série de valores estimados (por exemplo, esforço, custo, duração do projeto) e v_i são parâmetros independentes selecionados (por exemplo, LOC estimado ou FP).

Ferramentas automáticas de estimativa implementam uma ou mais técnicas de decomposição ou modelos empíricos e proporcionam uma opção atrativa para estimativas. Em tais sistemas, são descritos as características da organização de desenvolvimento (por exemplo, experiência, ambiente) e o software a ser desenvolvido. Com base nesses dados são estimados o custo e o esforço.

Cada uma dessas opções viáveis de estimativa de custo de software é apenas tão boa quanto os dados históricos usados para compor a estimativa. Se não existem dados históricos, os custos se baseiam em uma fundação muito precária. No Capítulo 25, examinamos as características de algumas das métricas de software que proporcionam a base para dados de estimativa histórica.

26.6 TÉCNICAS DE DECOMPOSIÇÃO

A estimativa de projeto de software é uma forma de solução de problema e, na maioria dos casos, o problema a ser resolvido (desenvolver uma estimativa de custo e esforço para um projeto de software) é muito complexo para ser considerado em uma única parte. Por essa razão, você deve decompor o problema, redefinindo-o como uma série de problemas menores (e, talvez, mais controláveis).

No Capítulo 24, a abordagem da decomposição foi discutida com base em dois pontos de vista: decomposição do problema e decomposição do processo. As estimativas usam uma ou ambas as formas de particionamento. Mas antes de fazer uma estimativa, entenda o escopo do software a ser criado e gere uma estimativa de seu “tamanho”.

26.6.1 Dimensionamento do software

A precisão de uma estimativa de projeto de software é baseada em vários itens:

- (1) o grau com que você estimou adequadamente o tamanho do produto a ser criado;
- (2) a habilidade para traduzir a estimativa de esforço humano, tempo de trabalho e recursos financeiros (uma função da disponibilidade de métricas de software confiáveis de projetos anteriores);
- (3) o grau com que o plano do projeto reflete as habilidades da equipe de software; e
- (4) a estabilidade dos requisitos do produto e o ambiente que suporta o trabalho de engenharia de software.

Nesta seção, consideraremos o problema do *dimensionamento do software*. Como a estimativa do projeto é apenas tão boa quanto a estimativa do trabalho a ser realizado, o dimensionamento representa seu primeiro grande desafio como planejador. No contexto do planejamento de projeto, tamanho refere-se a um resultado quantificável do projeto de software. Se for adotada uma abordagem direta, o tamanho pode ser medido em linhas de código (LOC). Se for escolhida uma abordagem indireta, o tamanho é representado como pontos de função (FP).

Putnam e Myers [Put92] sugerem quatro abordagens diferentes para o problema do dimensionamento:

- *Dimensionamento de “lógica nebulosa” (fuzzy logic)*. Essa abordagem usa as técnicas de raciocínio aproximado, a pedra fundamental da *lógica nebulosa*. Para utilizá-la, o planejador deve identificar o tipo de aplicação, estabelecer sua magnitude em uma escala qualitativa e então refinar a magnitude dentro do intervalo original.
- *Dimensionamento de pontos de função*. O planejador desenvolve estimativas das características do domínio da informação discutido no Capítulo 23.
- *Dimensionamento de componentes-padrão*. O software é composto por uma série de diferentes “componentes-padrão” que são genéricos em relação a determinada área de aplicação. Por exemplo, os componentes-padrão para um sistema de informações são

PONTO-CHAVE

O “tamanho” do software a ser criado pode ser estimado usando-se uma medida direta, LOC, ou uma medida indireta, FP.



Como dimensionamos o software que estamos planejando criar?

subsistemas, módulos, telas, relatórios, programas interativos, programas em lote, arquivos, LOC e instruções em nível de objeto. O planejador do projeto estima o número de ocorrências de cada componente-padrão e usa dados históricos de projeto para estimar o tamanho de cada um deles.

- *Dimensionamento de alteração.* Essa abordagem é empregada quando um projeto abrange o uso de software existente que deve ser modificado de alguma forma como parte do projeto. O planejador estima o número e tipo (por exemplo, reutilização, adição de código, mudança de código, exclusão de código) de modificações que devem ser feitas.

Putnam e Myers sugerem que os resultados de cada uma dessas abordagens de dimensionamento sejam combinados estatisticamente para criar uma estimativa de *três pontos* ou *valor esperado*. Isso é obtido desenvolvendo-se valores otimistas (baixo), mais prováveis e pessimistas (alto) para o tamanho e combinando-os por meio da Equação (26.1), descrita na Seção 26.6.2.

26.6.2 Estimativa baseada em problema

No Capítulo 25, linhas de código e pontos de função foram descritos como medidas por meio das quais podem ser calculadas as métricas de produtividade. Dados de LOC e FP são usados de duas maneiras durante a estimativa do projeto de software: (1) como variáveis de estimativa para “dimensionar” cada elemento do software e (2) como métricas de referência coletadas de projetos anteriores e utilizadas em conjunto com variáveis de estimativa para desenvolver projeções de custo e esforço.

Estimativas LOC e FP são técnicas distintas. No entanto, ambas têm muitas características em comum. Inicia-se com uma definição delimitada do escopo do software e daí tenta-se decompor a definição em funções de problemas que podem ser estimados individualmente. LOC ou FP (a variável de estimativa) é então estimada para cada função. Como alternativa, pode-se escolher um outro componente para dimensionamento como classes ou objetos, alterações ou processos de negócio afetados.

Métricas de produtividade de referência (por exemplo, LOC/pm ou FP/pm⁶) são aplicadas à variável apropriada de estimativa e, assim, se obtém o custo ou esforço para a função. As estimativas de função combinam-se para produzir uma estimativa geral para todo o projeto.

É importante observar, porém, que muitas vezes há uma dispersão substancial em métricas de produtividade para uma organização, não sendo aconselhável o uso de uma única métrica de produtividade de referência. Em geral, médias LOC/pm ou FP/pm deverão ser computadas por domínio de projeto. Os projetos deverão ser agrupados por tamanho de equipe, por área de aplicação, complexidade e outros parâmetros relevantes. Deverão ser calculadas as médias locais de domínio. Quando é estimado um novo projeto, esse deverá primeiro ser alocado a um domínio e, depois, à média de domínio apropriada para produtividade anterior deverá ser usada para gerar a estimativa.

As técnicas de estimativa LOC e FP diferem em nível de detalhe requerido para a decomposição e no alvo do particionamento. Quando é usada a LOC como variável de estimativa, a decomposição é absolutamente essencial e muitas vezes é adotada com níveis consideráveis de detalhes. Quanto maior o grau de particionamento, maior a probabilidade de serem desenvolvidas estimativas LOC razoavelmente precisas.

Para estimativas FP, a decomposição funciona de forma diferente. Em vez de focalizar-se na função, é estimada cada uma das características do domínio de informação – entradas, saídas, arquivos de dados, consultas e interfaces externas –, bem como os 14 valores de ajuste de complexidade discutidos no Capítulo 23. As estimativas resultantes podem então ser usadas para derivar um valor de FP que pode ser relacionado a dados anteriores e usado para gerar uma estimativa.

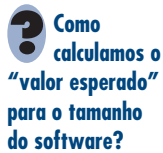
Independentemente da variável de estimativa utilizada, deve-se começar estimando um intervalo de valores para cada valor de função ou domínio de informação. Usando dados histó-

 **O que as estimativas baseadas em LOC e FP têm em comum?**



Ao coletar métricas de produtividade para projetos, estabeleça uma classificação de tipos de projeto. Isso permitirá o cálculo de médias específicas de domínio, tornando a estimativa mais precisa.

6 A abreviatura *pm* significa pessoa-mês de trabalho.



ricos ou (quando tudo o mais falhar) intuição, estimam-se valores de tamanho otimista, mais provável e pessimista para cada função ou contagem para cada valor do domínio de informações. Quando um intervalo de valores é especificado, é fornecida uma indicação implícita do grau de incerteza.

Um valor esperado ou de três pontos pode então ser calculado. O *valor esperado* para a variável de estimativa (tamanho) S pode ser calculado como uma média ponderada das estimativas otimista (S_{opt}), mais provável (S_m) e pessimista (S_{pess}). Por exemplo,

$$S = \frac{S_{opt} + 4S_m + S_{pess}}{6} \quad (26.1)$$

oferece a maior credibilidade da estimativa “mais provável” e segue uma distribuição beta de probabilidade. Assumimos que há uma probabilidade muito pequena de que o tamanho real fique fora dos valores otimista e pessimista.

Uma vez determinado o valor esperado para a variável de estimativa, aplicam-se os dados de produtividade histórica de LOC ou FP. As estimativas estão corretas? A única resposta razoável a essa pergunta é: “Não se pode ter certeza”. Qualquer técnica de estimativa, não importa quão sofisticada seja, deve passar por uma verificação cruzada com outra abordagem. Mesmo assim, o bom senso e a experiência devem prevalecer.

26.6.3 Um exemplo de estimativa baseada em LOC

Como exemplo das técnicas de estimativa em LOC e FP baseadas em problema, vamos considerar um pacote de software a ser desenvolvido para uma aplicação de projeto auxiliado por computador para componentes mecânicos. O software deve ser executado em uma estação de trabalho de engenharia e deve ter interface com vários periféricos gráficos incluindo um mouse, um teclado, um monitor colorido de alta resolução e uma impressora laser. Pode ser formulada uma definição preliminar do escopo do software:



Muitas aplicações modernas residem em uma rede ou fazem parte de uma arquitetura cliente-servidor. Portanto, certifique-se de que suas estimativas incluem a mão de obra requerida para desenvolver software de “infraestrutura”.

O software CAD mecânico aceitará dados geométricos bidimensionais e tridimensionais fornecidos por um engenheiro. O engenheiro vai interagir e controlar o sistema CAD através de uma interface de usuário que irá exibir características de um bom design de interface homem/máquina. Todos os dados geométricos e outras informações de suporte serão mantidos em uma base de dados CAD. Serão desenvolvidos módulos de análise de projeto para produzir a saída requerida, a ser exibida em uma variedade de dispositivos gráficos. O software será projetado para controlar e interagir com dispositivos periféricos que incluem um mouse, um teclado, impressora laser e um plotter.

Essa definição de escopo é preliminar – não é delimitada. Cada sentença deverá ser expandida para proporcionar detalhe concreto e limites quantitativos. Por exemplo, antes de iniciar a estimativa, o planejador deve determinar o que significa “características de projeto de uma boa interface homem/máquina” ou qual deverão ser o tamanho e a sofisticação da “base de dados CAD”.

Para nossos propósitos, suponha que ocorreu um refinamento adicional e que as principais funções do software listadas na Figura 26.2 estão identificadas. Seguindo a técnica de decomposição para LOC, é desenvolvida uma tabela de estimativas (Figura 26.2). Para cada função é desenvolvido um intervalo de estimativas LOC. Por exemplo, o intervalo de estimativas LOC para a função de análise geométrica 3-D é otimista, 4.600 LOC; mais provável, 6.900 LOC; e pessimista, 8.600 LOC. Aplicando a Equação 26.1, o valor esperado para a função de análise geométrica 3D é 6.800 LOC. Outras estimativas são obtidas de forma semelhante. Somando verticalmente na coluna de estimativa LOC, obtém-se uma estimativa de 33.200 linhas de código para o sistema CAD.

Um exame de dados históricos indica que a produtividade média organizacional para sistemas desse tipo é de 620 LOC/pm. Com base em um valor bruto da mão de obra de \$ 8 mil por mês, o custo por linha de código é de aproximadamente \$ 13. Com base na estimativa LOC e dados históricos de produtividade, o custo total estimado do projeto é de \$ 431 mil e o esforço estimado é de 54 pessoas-mês.⁷



Não ceda à tentação de usar esse resultado como sua estimativa de projeto. Você deverá derivar um outro resultado usando uma abordagem diferente.

⁷ As estimativas são arredondadas para o próximo \$ 1.000 e pessoa-mês. Precisão maior não é necessária nem realística, dadas as limitações da precisão da estimativa.

FIGURA 26.2

Tabela de estimativas para o método LOC

Função	LOC estimado
Interface de usuário e recurso de controle	2.300
Análise geométrica bidimensional	5.300
Análise geométrica tridimensional	6.800
Gerenciamento de base de dados	3.350
Recursos de visualização da computação gráfica	4.950
Função de controle de periféricos	2.100
Módulos de análise do projeto	8.400
<i>Linhas de código estimadas</i>	<i>33.200</i>

CASASEGURA



Estimativa

Cena: Escritório de Doug Miller no início do planejamento.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*), Vinod Raman, Jamie Lazar e outros membros da equipe de engenharia de software.

Conversa:

Doug: Precisamos desenvolver uma estimativa de esforço para projeto e depois temos de definir um microcronograma para o primeiro incremento e um macrocronograma para os demais incrementos.

Vinod (acenando afirmativamente): Ok, mas não definimos nenhum incremento ainda.

Doug: Sim, mas é por isso que precisamos fazer estimativas.

Jamie (contrariado): Você quer saber quanto tempo vamos gastar para fazer isso?

Doug: Veja o que eu preciso. Primeiro, precisamos decompor funcionalmente o software *CasaSegura* em alto nível... Depois temos de estimar o número de linhas de código que cada função terá... Depois...

Jamie: Espera um pouco! Como você acha que vamos fazer isso?

Vinod: Já fizemos em projetos anteriores. Você começa com casos de uso, determina a funcionalidade requerida para implementar cada um, arrisca um palpite quanto ao número de LOC para cada parte da função. A melhor abordagem é que todos façam isso de forma independente e depois comparemos os resultados.

Doug: Ou você pode fazer uma decomposição funcional do projeto inteiro.

Jamie: Mas isso vai demorar uma eternidade e precisamos começar.

Vinod: Não... Isso pode ser feito em algumas horas... Na verdade, nesta manhã.

Doug: Concorde... Não podemos esperar exatidão, apenas uma vaga ideia do tamanho que o *CasaSegura* terá.

Jamie: Acho que deveríamos apenas estimar o esforço... Só isso.

Doug: Faremos isso também. Depois usaremos ambas as estimativas como verificação cruzada.

Vinod: Então vamos fazer...

26.6.4 Um exemplo de estimativa baseada em FP

A decomposição para estimativa baseada em FP concentra-se em valores do domínio da informação em lugar das funções de software. De acordo com a tabela apresentada na Figura 26.3, você estimaria entradas, saídas, consultas, arquivos e interfaces externas para o software CAD. É computado um valor de FP usando a técnica discutida no Capítulo 23. Para os propósitos dessa estimativa, o fator de peso da complexidade é adotado como médio. A Figura 26.3 apresenta os resultados dessa estimativa.

FIGURA 26.3

Estimando valores do domínio de informações

Valor do domínio	Esti- Saídas	Estimativa Consulta	Estimativa computada	FB Peso computado		
Número de entradas externas	20	24	30	24	4	97
Número de saídas externas	12	15	22	16	5	78
Número de consultas externas	16	22	28	22	5	88
Número de arquivos lógico internos	4	4	5	4	10	42
Número de arquivos de interface externos	2	2	3	2	7	15
Contagem total						320

É estimado cada um dos fatores de peso da complexidade; o fator de ajuste do valor é calculado conforme descrito no Capítulo 23:

Fator	Valor
Backup e recuperação	4
Comunicações de dados	2
Processamento distribuído	0
Desempenho crítico	4
Ambiente operacional existente	3
Entrada de dados on-line	4
Transações de entrada em múltiplas telas	5
Arquivos mestres atualizados on-line	3
Complexidade dos valores dos domínios de informação	5
Complexidade do processamento interno	5
Código projetado para reutilização	4
Conversão/instalação no projeto	3
Instalações múltiplas	5
Aplicação projetada para alteração	5
Fator de ajuste de valor	1,17

Por fim, é obtido o número estimado de FP:

$$FP_{\text{estimado}} = \text{contagem total} \times [0,65 + 0,01 \times \Sigma(F_i)] = 375$$

A produtividade média organizacional para sistemas desse tipo é de 6,5 FP/pm. Com base em um valor bruto de mão de obra de \$ 8 mil por mês, o custo por FP é aproximadamente de \$ 1.230. Com base na estimativa FP e nos dados de produtividade histórica, o custo total estimado do projeto é de \$ 461 mil, e a estimativa de esforço é de 58 pessoas-mês.

26.6.5 Estimativas baseadas em processo

A técnica mais comum para estimativa de projeto é baseá-la no processo a ser usado. Isto é, o processo é decomposto em um conjunto relativamente pequeno de tarefas, e é estimado o trabalho necessário para executar cada tarefa.

FIGURA 26.4

Tabela de estimativa baseada em processo

Atividade →	CC	Planejamento	Análise de risco	Engenharia		Liberação da versão		AC	Totais
Tarefa →				Análises	Design	Code	Test		
Função									
Y									
UICF				0,50	2,50	0,40	5,00	n/a	8,40
2DGA				0,75	4,00	0,60	2,00	n/a	7,35
3DGA				0,50	4,00	1,00	3,00	n/a	8,50
CGDF				0,50	3,00	1,00	1,50	n/a	6,00
DBM				0,50	3,00	0,75	1,50	n/a	5,75
PCF				0,25	2,00	0,50	1,50	n/a	4,25
DAM				0,50	2,00	0,50	2,00	n/a	5,00
Totais	0,25	0,25	0,25	3,50	20,50	4,50	16,50		46,00
% de mão de obra	1%	1%	1%	8%	45%	10%	36%		

CC = comunicação com o cliente AC = avaliação pelo cliente

Assim como as técnicas baseadas em problemas, a estimativa baseada em processo começa com um delineamento das funções de software obtidas do escopo do projeto. Para cada função, deve ser executada uma série de atividades estruturais. As funções e atividades estruturais relacionadas⁸ podem ser representadas como parte de uma tabela similar à da Figura 26.4.



Se houver tempo suficiente, use maior detalhamento ao especificar tarefas na Figura 26.4. Por exemplo, subdivida a análise em suas tarefas principais e estime cada uma separadamente.

Uma vez combinadas as funções do problema e atividades de processo, você pode estimar o esforço (por exemplo, pessoas-mês) necessário para executar cada atividade do processo de software, para cada função de software. Esses dados constituem a matriz central da tabela da Figura 26.4. São então aplicados os valores médios de preço de mão de obra (isto é, custo/unidade de esforço) ao esforço estimado para cada atividade de processo. É muito provável que os valores da mão de obra irão variar para cada tarefa. O pessoal mais experiente se envolve mais intensamente nas primeiras atividades estruturais e geralmente são mais caros do que o pessoal menos experiente envolvido na construção e liberação das versões.

Custos e esforço para cada função e para cada atividade estrutural são calculados, como última etapa. Se a estimativa baseada em processo é feita independentemente da estimativa baseada em LOC ou FP, temos agora duas ou três estimativas para custo e esforço que podem ser comparadas e reconciliadas. Se ambos os conjuntos de estimativas apresentarem concordância razoável, há boas razões para acreditar que as estimativas são confiáveis. Por outro lado, se os resultados dessas técnicas de decomposição mostrarem pouca concordância, deve ser executada uma investigação e análise mais profunda.

26.6.6 Um exemplo de estimativa baseada em processo

Para ilustrar o uso de estimativas baseadas em processo, considere o software CAD introduzido na Seção 26.6.3. A configuração do sistema e todas as funções de software permanecem inalteradas e são indicadas pelo escopo do projeto.

De acordo com a tabela baseada em processo mostrada na Figura 26.4, são feitas as estimativas de esforço (em pessoas-mês) para cada atividade de engenharia de software para cada função do software CAD (abreviado para maior simplicidade). As atividades de engenharia e construção subdividem-se nas principais tarefas de engenharia de software da tabela. São feitas estimativas aproximadas de esforço para comunicação com o cliente, planejamento e análise de riscos. Os totais são colocados na parte inferior da tabela. Os totais horizontais e verticais fornecem uma indicação do esforço estimado necessário para análise, projeto, codificação e

“É melhor entender os fundamentos de uma estimativa antes de utilizá-la.”

**Barry Boehm e
Richard Fairley**

⁸ As atividades estruturais escolhidas para este projeto diferem um pouco das atividades genéricas discutidas no Capítulo 2. São elas: comunicação com o cliente (*customer communication* – CC), planejamento, análise de risco, engenharia e construção/liberação das versões.

teste. Devemos observar que 53% de todo o esforço são gastos nas tarefas de engenharia inicial (análise de requisitos e projeto), indicando a importância relativa deste trabalho.

Com base na taxa média bruta de mão de obra de \$ 8 mil por mês, o custo total estimado do projeto é de \$ 368 mil e o esforço estimado é de 46 pessoas-mês. Se você quiser, as taxas de mão de obra podem ser associadas a cada atividade estrutural ou a cada tarefa de engenharia de software e calculadas separadamente.

26.6.7 Estimativa com casos de uso

Conforme afirmamos na Parte 2 deste livro, os casos de uso proporcionam a uma equipe de software informações sobre o escopo do software e os requisitos. No entanto, é problemático desenvolver uma abordagem de estimativa com casos de uso pelas seguintes razões [Smi99]:

- Casos de uso são descritos por meio de muitos formatos e estilos diferentes – não há um formato-padrão.
- Casos de uso representam uma visão externa (a visão do usuário) do software e podem, portanto, ser escritos em muitos diferentes níveis de abstração.
- Casos de uso não tratam da complexidade das funções e características que são descritas.
- Casos de uso podem descrever comportamento complexo (por exemplo, interações) que envolvem muitas funções e características.

Diferentemente de uma LOC ou ponto de função, um “caso de uso” pode precisar de meses de trabalho enquanto o caso de uso de outra pessoa pode ser implementado em um dia ou dois.

Embora muitos pesquisadores tenham considerado os casos de uso uma informação de estimativa, até agora não existe nenhum método de estimativa comprovado.⁹ Smith [Smi99] sugere que os casos de uso podem ser empregados para estimativa, mas somente se forem considerados no contexto da “hierarquia estrutural” em que são usados para descrever.

Smith afirma que qualquer nível dessa hierarquia estrutural pode ser descrito por não mais do que 10 casos de uso. Cada um poderia abranger não mais do que trinta cenários distintos. Obviamente, casos de uso que descrevem um grande sistema são escritos com um nível muito mais alto de abstração (e representam consideravelmente bem mais esforço de desenvolvimento) do que aqueles escritos para descrever um único subsistema. Portanto, antes que os casos de uso possam ser empregados para estimativas, é estabelecido o nível da hierarquia estrutural, determinado o tamanho médio (em páginas) de cada caso, definido o tipo de software (por exemplo, em tempo real, corporativo, engenharia/científico, WebApp, embarcado), e é considerada uma arquitetura primitiva para o sistema. Uma vez estabelecidas essas características, podem ser utilizados dados empíricos para estabelecer o número estimado de LOC ou FP por caso de uso (para cada nível de hierarquia). Usam-se então dados históricos para calcular o esforço necessário para desenvolver o sistema.

Para ilustrar como essa computação pode ser feita, considere a seguinte relação:¹⁰

$$\text{LOC estimado} = N \times \text{LOC}_{\text{média}} + [(S_d/S_h - 1) + (P_d/P_h - 1)] \times \text{LOC}_{\text{ajustado}} \quad (26.2)$$

em que

- N = número atual de casos de uso
- $\text{LOC}_{\text{média}}$ = média histórica de LOC por caso de uso para esse tipo de subsistema
- $\text{LOC}_{\text{ajustado}}$ = representa um ajuste baseado em n por cento de $\text{LOC}_{\text{média}}$ em que n é definido localmente e representa a diferença entre esse projeto e a “média” dos outros projetos

⁹ Trabalho recente sobre a derivação de *pontos de caso de uso* [Cle06] pode finalmente levar a uma abordagem prática de estimativa por meio de casos de uso.

¹⁰ É importante observar que a Expressão (26.2) é usada apenas para fins de ilustração. Assim como todos os modelos de estimativa, ela pode ser validada localmente para que possa ser usada com segurança.

? Por que é difícil desenvolver uma técnica de estimativa usando casos de uso?

- S_a = cenários atuais por caso de uso
 S_h = cenários médios por caso de uso para esse tipo de subsistema
 P_a = número atual de páginas por caso de uso
 P_h = número médio de páginas por caso de uso para esse tipo de subsistema

A Expressão (26.2) poderia ser usada para desenvolver uma estimativa aproximada do número de LOC com base no número real de casos de uso ajustado pelo número de cenários e tamanhos de página dos casos. O ajuste representa até n por cento da média histórica de LOC por caso de uso.

26.6.8 Exemplo de estimativa baseada em caso de uso

O software CAD apresentado na Seção 26.6.3 é composto de três grupos de subsistemas: subsistema da interface do usuário (inclui UICF), grupo do subsistema de engenharia (inclui os subsistemas 2DGA, 3DGA e DAM) e grupo do subsistema de infraestrutura (inclui os subsistemas CGDF e PCF). Seis casos de uso descrevem o subsistema de interface de usuário. Cada caso é descrito por não mais do que 10 cenários e tem um tamanho médio de 6 páginas. O grupo do subsistema de engenharia é descrito por 10 casos de uso (considerados pertencentes a um nível mais alto na hierarquia estrutural). Cada um deles tem não mais de 20 cenários associados e um tamanho médio de 8 páginas. Por fim, o grupo do subsistema infraestrutura é descrito por 5 casos de uso com uma média de apenas seis cenários e um tamanho médio de 5 páginas.

Por meio das relações descritas na Expressão (26.2) com $n = 30\%$, calcula-se a tabela da Figura 26.5. Considerando a primeira linha da tabela, dados históricos indicam que o software da interface de usuário (UI) requer uma média de 800 LOC por caso de uso quando este não tiver mais do que 12 cenários e for descrito em menos de 5 páginas. Esses dados se adaptam razoavelmente bem para o sistema CAD. Portanto, a estimativa LOC para o subsistema de interface de usuário é calculado usando-se a Expressão (26.2). Pela mesma abordagem, são feitas estimativas para os grupos de subsistemas de engenharia e infraestrutura. A Figura 26.5 resume as estimativas e indica que o tamanho total do CAD é estimado em 42.500 linhas de código (LOC).

Usando 620 LOC/pm como produtividade média para sistemas desse tipo e um custo bruto de mão de obra de \$ 8 mil por mês, o custo por linha de código é de aproximadamente \$ 13. Com base na estimativa de casos de uso e dados de produtividade histórica, o custo total estimado do projeto é de \$ 552 mil e o esforço estimado é de 68 pessoas-mês.

FIGURA 26.5

Estimativa de caso de uso

	casos de uso	cenários	páginas	cenários	páginas	LOC	LOC estimado
Subsistema de interface de usuário	6	10	6	12	5	560	3.366
Grupo do subsistema de engenharia	10	20	8	16	8	3100	31.233
Grupo do subsistema de infraestrutura	5	6	5	10	6	1650	7.970
Total estimado de LOC							42.568

"Métodos complicados podem não produzir uma estimativa exata, particularmente quando os desenvolvedores podem incorporar sua própria intuição na estimativa."

Philip Johnson e outros

26.6.9 Reconciliando estimativas

As técnicas de estimativa discutidas nas seções anteriores resultam em múltiplas estimativas que devem ser reconciliadas para produzir uma única estimativa de esforço, duração de projeto ou custo. Para tanto, vamos considerar novamente o software CAD introduzido na Seção 26.6.3.

O esforço total estimado para o software CAD varia desde um valor baixo de 46 pessoas-mês (derivado de uma abordagem de estimativa baseada em processo) até um valor alto de 68 pessoas-mês (derivado de uma estimativa de caso de uso). A estimativa média (com as quatro abordagens) é de 56 pessoas-mês. A variação em relação à estimativa média é de aproximadamente 18% para baixo e 21% para cima.

O que acontece quando a concordância entre as estimativas é baixa? A resposta a essa questão requer uma reavaliação das informações usadas para fazer as estimativas. Estimativas que divergem muito podem ser atribuídas a uma de duas causas: (1) o escopo do projeto não é entendido adequadamente ou foi mal interpretado pelo planejador ou (2) os dados de produtividade usados para as técnicas de estimativa baseada em problema são inadequados para a aplicação, obsoletos (porque não refletem mais com precisão a organização de engenharia de software) ou foram mal aplicados. Você deverá determinar a causa da divergência e reconciliar as estimativas.



Técnicas automáticas de estimativa para projetos de software

As ferramentas de estimativa automáticas permitem ao planejador estimar o custo e o esforço e executar análises do tipo “e se...” para variáveis importantes do projeto como data de entrega ou equipe. Embora existam muitas ferramentas de estimativas automáticas, todas apresentam as mesmas características gerais e executam as seis funções genéricas a seguir [Jon96]:

1. *Dimensionamento dos entregáveis do projeto.* O “tamanho” de um ou mais artefatos de software é estimado. Artefatos incluem a representação externa do software (por exemplo, tela, relatórios), o próprio software (por exemplo, KLOC), funcionalidade fornecida (por exemplo, pontos de função) e informações descritivas (por exemplo, documentos).
2. *Seleção de atividades de projeto.* É selecionada a estrutura apropriada de processo e especificado o conjunto de tarefas de engenharia de software.
3. *Previsão do número de pessoas.* É especificado o número de pessoas que estarão disponíveis para o trabalho. Esse é um dado importante porque a relação entre pessoas disponíveis e trabalho (esforço previsto) é altamente não linear.

4. *Previsão do esforço de software.* As ferramentas de estimativa empregam um ou mais modelos (Seção 26.7) que relacionam o tamanho dos entregáveis do projeto com o esforço necessário para produzi-los.

5. *Previsão do custo do software.* Dados os resultados da etapa 4, os custos podem ser calculados alocando-se valores da mão de obra às atividades de projeto citadas na etapa 2.

6. *Previsão dos cronogramas do software.* Ao conhecermos esforço, número de pessoas e atividades de projeto, podemos produzir um esboço de cronograma alocando mão de obra às atividades de engenharia de software com base nos modelos recomendados para distribuição do esforço discutidos mais adiante neste capítulo.

Quando são aplicadas diferentes ferramentas de estimativa para os mesmos dados de projeto, pode ser encontrada uma variação razoavelmente grande nos resultados estimados. Mais importante ainda, os valores previstos muitas vezes são significativamente diferentes dos valores reais. Isso reforça a noção de que o produto das ferramentas de estimativas deverá ser usado como “ponto de dados” por meio do qual são obtidas as estimativas – não como a única fonte para uma estimativa.

INFORMAÇÕES

26.7 MODELOS EMPÍRICOS DE ESTIMATIVA

Um modelo de estimativa para software usa fórmulas derivadas empiricamente para prever o esforço como uma função de LOC ou FP.¹¹ Valores para LOC ou FP são estimados usando a abordagem descrita nas Seções 26.6.3 e 26.6.4. Em vez de utilizarmos as tabelas descritas naquelas seções, os valores resultantes para LOC ou FP são anexados ao modelo de estimativa.

Os dados empíricos que suportam muitos modelos de estimativa são obtidos de uma amostragem limitada de projetos. Por essa razão, nenhum modelo de estimativa é apropriado para todas as classes de software e todos os ambientes de desenvolvimento. Portanto, você deverá usar os resultados obtidos desses modelos com muito critério.

Um modelo de estimativa deverá ser calibrado para refletir condições locais. O modelo deverá ser testado aplicando-se dados coletados de projetos completos, anexando os dados no modelo e comparando resultados reais com os previstos. Se a concordância for baixa, o modelo deve ser ajustado e retestado antes de ser usado.

PONTO-CHAVE

Um modelo de estimativa reflete a população de projetos com base na qual foi obtido. Portanto, o modelo é sensível ao domínio.

¹¹ Um modelo empírico que utiliza casos de uso como variável independente é sugerido na Seção 26.6.6. No entanto, apenas alguns apareceram na literatura até agora.

26.7.1 Estrutura dos modelos de estimativa

Um modelo típico de estimativa é obtido por meio da análise de regressão sobre dados coletados de projetos de software anteriores. A estrutura geral desses modelos toma a forma [Mat94]

$$E = A + B \times (e_v)^c \quad (26.3)$$

em que A , B e C são constantes obtidas empiricamente, E é o esforço em pessoas-mês e e_v é a variável de estimativa (LOC ou FP). Além da relação descrita na Equação (26.3), a maioria dos modelos de estimativa tem alguma forma de componente de ajuste de projeto que permite que E seja ajustado por outras características do projeto (por exemplo, complexidade do problema, experiência da equipe, ambiente de desenvolvimento). Entre os muitos modelos de estimativas orientados para LOC propostos na literatura estão



Nenhum desses modelos deverá ser usado sem uma cuidadosa calibração com o seu ambiente.

$$E = 5,2 \times (\text{KLOC})^{0,91}$$

Modelo Walston-Felix

$$E = 5,5 + 0,73 \times (\text{KLOC})^{1,16}$$

Modelo Bailey-Basili

$$E = 3,2 \times (\text{KLOC})^{1,05}$$

Modelo Boehm simples

$$E = 5,288 \times (\text{KLOC})^{1,047}$$

Modelo Doty para KLOC > 9

Também foram propostos modelos orientados a FP. São eles

$$E = -91,4 + 0,355 \text{ FP}$$

Modelo Albrecht e Gaffney

$$E = -37 + 0,96 \text{ FP}$$

Modelo Kemerer

$$E = -12,88 + 0,405 \text{ FP}$$

Modelo de regressão de pequeno projeto

Um rápido exame desses modelos indica que cada um produzirá um resultado diferente para os mesmos valores de LOC ou FP. A implicação é clara. Modelos de estimativa devem ser calibrados para as necessidades locais!

26.7.2 O modelo COCOMO II

WebRef

Informações detalhadas sobre COCOMO II, incluindo software para download, podem ser obtidas no endereço sunset.usc.edu/research/COCOMOII/cocomo_main.html.

Em seu livro clássico sobre “economia da engenharia de software” (*software engineering economics*), Barry Boehm [Boe81] introduziu uma hierarquia de modelos de estimativa de software com o nome COCOMO, que significa *CONstructive COSt MODEL* (modelo construtivo de custo). O COCOMO original tornou-se um dos modelos de estimativa de custo de software mais amplamente usado e discutido na indústria. Ele evoluiu para um modelo de estimativa mais abrangente, chamado COCOMOII [Boe00]. Assim como seu predecessor, o COCOMO II é na realidade uma hierarquia de modelos de estimativa que trata das seguintes áreas:

- *Modelo de composição de aplicação.* Usado durante os primeiros estágios da engenharia de software, em que protótipo das interfaces de usuário, a consideração da interação de software e sistema, a avaliação do desempenho e a avaliação da maturidade da tecnologia são de suma importância.
- *Modelo de estágio do início do projeto.* Usado quando os requisitos tiverem sido estabilizados e a arquitetura básica de software tiver sido estabelecida.
- *Modelo de estágio pós-arquitetura.* Usado durante a construção do software.

Assim como todos os modelos de estimativa para software, os modelos COCOMO II requerem informações de tamanho. Há disponíveis três diferentes opções como parte da hierarquia de modelo: pontos de objeto, pontos de função e linhas de código-fonte.

O modelo de composição de aplicação COCOMO II utiliza pontos de objeto e é ilustrado nos próximos parágrafos. Devemos observar que há também disponíveis outros modelos de estimativa mais sofisticados (usando FP e KLOC) como parte do COCOMO II.

Assim como os pontos de função, o *ponto de objeto* é uma medida indireta de software calculada por meio de contagens dos números de (1) telas (na interface do usuário), (2) relatórios e



FIGURA 26.6

Peso da complexidade para tipos de objeto

Fonte: (Boe96)

Tipo de objeto	Peso da complexidade		
	Simple	Médio	Difícil
Tela	1	2	3
Relatório	2	5	8
Componente 3GL			10

(3) componentes que podem ser necessários para construir a aplicação. Cada instância de objeto (por exemplo, uma tela ou um relatório) é classificada em um dentre três níveis de complexidade (simple, médio ou difícil) usando critérios sugeridos por Boehm [Boe96]. Essencialmente, a complexidade é uma função da quantidade e origem das tabelas de dados-cliente e servidor necessárias para gerar a tela ou relatório e o número de visualizações ou seções apresentadas como parte da tela ou relatório.

Uma vez determinada a complexidade, os números de telas, relatórios e componentes são ponderados de acordo com a tabela da Figura 26.6. A contagem de pontos de objeto é então determinada multiplicando-se o número original de instâncias de objeto pelo fator de peso na figura e somando para obter o total da contagem de pontos de objeto. Quando deve ser aplicado desenvolvimento baseado em componente ou reutilização de software em geral, é estimada a porcentagem de reutilização (% reuso) e é ajustada a contagem de pontos de objeto:

$$NOP = (\text{pontos de objeto}) \times [(100 - \% \text{reuso})/100]$$

em que NOP é definido como novos pontos de objeto.

Para derivar uma estimativa de esforço com base no valor calculado para NOP, deve ser derivada uma “taxa de produtividade”. A Figura 26.7 apresenta a taxa de produtividade

$$PROD = \frac{NOP}{\text{pessoa-mês}}$$

para diferentes níveis de experiência do desenvolvedor e maturidade do ambiente de desenvolvimento.

Uma vez determinada a taxa de produtividade, calcula-se a estimativa de esforço do projeto usando

$$\text{Esforço estimado} = \frac{NOP}{PROD}$$

Em modelos COCOMO II mais avançados,¹² é necessária uma variedade de fatores de escala, custos e procedimentos de ajuste. Uma discussão completa desses tópicos está além do escopo deste livro. Se você estiver interessado, veja [Boe00] ou visite o site do COCOMO II.

FIGURA 26.7

Taxa de produtividade para pontos de objeto

Fonte: (Boe96)

Experiência/capacidade do desenvolvedor	Muito baixa	Baixa	Nominal	Alta	Muito alta
Maturidade/capacidade do ambiente	Muito baixa	Baixa	Nominal	Alta	Muito alta
PROD	4	7	13	25	50

¹² Conforme afirmamos anteriormente, esses modelos usam contagens FP e KLOC para a variável tamanho.

26.7.3 A Equação do software

A *equação do software* [Put92] é um modelo dinâmico multivariável que assume uma distribuição específica de esforço durante a vida de um projeto de desenvolvimento de software. O modelo foi derivado dos dados de produtividade coletados em mais de 4 mil projetos de software. Com base nesses dados, derivamos um modelo de estimativa da forma

$$E = \frac{\text{LOC} \times B^{0,333}}{P^3} \times \frac{1}{t^4} \quad (26.4)$$

em que

E = esforço em pessoas-mês ou pessoas-ano

t = duração do projeto em meses ou anos

B = “fator de habilidades especiais”¹³

P = “parâmetro de produtividade” que reflete: maturidade geral do processo e práticas gerenciais, a amplitude na qual são usadas as boas práticas de engenharia de software, o nível de linguagens de programação usado, o estado do ambiente de software, as habilidades e experiência da equipe de software e a complexidade da aplicação

Os valores típicos podem ser $P = 2$ mil para desenvolvimento de software embutido em tempo real, $P = 10$ mil para software de telecomunicações e sistemas, e $P = 28$ mil para aplicações de sistemas comerciais. O parâmetro de produtividade pode ser derivado para condições locais por meio de dados históricos coletados de trabalhos de desenvolvimento executados no passado.

Você deve observar que a equação de software tem dois parâmetros independentes: (1) uma estimativa do tamanho (em LOC) e (2) uma indicação da duração do projeto em meses corridos ou anos.

Para simplificarem o processo de estimativa e usarem uma forma mais comum para o modelo de estimativa, Putnam e Myers [Put92] sugerem uma série de equações derivadas da equação do software. O tempo de desenvolvimento mínimo é definido como

$$t_{\min} = 8,14 \frac{\text{LOC}}{P^{0,43}} \text{ em meses para } t_{\min} > 6 \text{ meses} \quad (26.5a)$$

$$E = 180 B t^3 \text{ em pessoas-mês para } E \geq 20 \text{ pessoas-mês} \quad (26.5b)$$

Note que t na Equação (26.5b) está representado em anos.

Usando a Equação (26.5) com $P = 12.000$ (o valor recomendado para software científico) para o software CAD discutido anteriormente neste capítulo,

$$t_{\min} = 8,14 \times \frac{33.200}{12.000^{0,43}} = 12,6 \text{ meses corridos}$$

$$E = 180 \times 0,28 \times (1,05)^3 = 58 \text{ pessoas-mês}$$

Os resultados da equação do software correspondem favoravelmente com a estimativa desenvolvida na Seção 26.6. Assim como o modelo COCOMO discutido na Seção 26.7.2, a equação do software continua evoluindo. Uma discussão mais detalhada dessa abordagem de estimativa pode ser encontrada em [Put97b].

WebRef

Informações sobre ferramentas de estimativas de custo de software originadas da equação do software podem ser encontradas em www.qsm.com.

¹³ B aumenta lentamente à medida que cresce a “necessidade de integração, teste, garantia de qualidade, documentação e habilidades de gerenciamento” [Put92]. Para pequenos programas (KLOC = 5 a 15), $B = 0,16$. Para programas maiores do que 70 KLOC, $B = 0,39$.

26.8 ESTIMATIVA PARA PROJETOS ORIENTADOS A OBJETO

É conveniente suplementar os métodos convencionais de estimativa de custo de software com uma técnica criada explicitamente para software orientado a objeto. Lorenz e Kidd [Lor94] sugerem a seguinte abordagem:

1. Desenvolva estimativas usando decomposição de esforço, análise FP e qualquer outro método para aplicações convencionais.
2. Usando o modelo de requisitos (Capítulo 6), desenvolva casos de uso e determine uma contagem. Reconheça que o número de casos de uso pode mudar à medida que o projeto avança.
3. Com base no modelo de requisitos, determine o número de classes-chave (chamadas de classes de análise no Capítulo 6).
4. Classifique o tipo de interface para a aplicação e desenvolva um multiplicador para classes de apoio:

Tipo de Interface	Multiplicador
Sem GUI	2,0
Interface de usuário baseada em texto	2,25
GUI	2,5
GUI Complexa	3,0

Multiplique o número de classes-chave (passo 3) pelo multiplicador para obter uma estimativa do número de classes de apoio.

5. Multiplique o número total de classes (classes-chave + classes de apoio) pelo número médio de unidades de trabalho por classe. Lorenz e Kidd sugerem 15 a 20 pessoas-dia por classe.
6. Faça uma verificação cruzada da estimativa baseada em classe multiplicando o número médio de unidades de trabalho por caso de uso.

26.9 TÉCNICAS ESPECIALIZADAS DE ESTIMATIVA

As técnicas de estimativa discutidas nas Seções 26.6 a 26.8 podem ser empregadas em qualquer projeto de software. No entanto, quando uma equipe de software encontra um prazo de projeto extremamente curto (semanas em vez de meses), no qual é provável que haja uma sequência contínua de alterações, o planejamento do projeto em geral e a estimativa em particular deverão ser abreviados.¹⁴ Nas próximas seções, examinaremos duas técnicas especializadas de estimativa.

26.9.1 Estimativa para desenvolvimento ágil

Como os requisitos para um projeto ágil (Capítulo 3) são definidos por uma série de cenários de usuário (por exemplo, “histórias” em Programação Extrema), é possível desenvolver uma abordagem de estimativa informal, razoavelmente disciplinada e significativa no contexto de planejamento de projeto para cada incremento de software. A estimativa para projetos ágeis usa uma abordagem de decomposição que abrange os seguintes passos:

1. Cada cenário de usuário (o equivalente a um minicaso de uso criado bem no início de um projeto por usuários finais ou outros interessados) é considerado separadamente para fins de estimativa.

 Como as estimativas são desenvolvidas quando aplicado um processo ágil?

¹⁴ “Abreviado” não significa eliminado. Mesmo os projetos de curta duração deverão ser planejados, e a estimativa é o fundamento do planejamento.

PONTO-CHAVE

No contexto de estimativa para projetos ágeis, “volume” é uma estimativa do tamanho total de um cenário de usuário em LOC ou FP.

2. O cenário é decomposto em uma série de tarefas de engenharia de software que serão necessárias para desenvolvê-lo.
- 3a. O esforço necessário para cada tarefa é estimado separadamente. Nota: A estimativa pode ser baseada em dados históricos, em um modelo empírico ou na “experiência”.
- 3b. Como alternativa, o “volume” do cenário pode ser estimado em LOC, FP ou alguma outra medida orientada para volume (por exemplo, contagem de caso de uso).
- 4a. As estimativas para cada tarefa são somadas para criar uma estimativa para o cenário.
- 4b. Como alternativa, o volume estimado para o cenário é traduzido em esforço usando dados históricos.
5. As estimativas de esforço para todos os cenários implementados para determinado incremento de software são somadas para desenvolver a estimativa de esforço para o incremento.

Como a duração necessária para o desenvolvimento de um incremento de software é bastante curta (tipicamente três a seis semanas), essa abordagem de estimativa serve a duas finalidades: (1) para a certificação de que o número de cenários a ser incluídos no incremento corresponde aos recursos disponíveis, e (2) para estabelecer uma base para alocar esforço na medida em que o incremento é desenvolvido.

26.9.2 Estimativa para projetos para WebApp

Os projetos para WebApp muitas vezes adotam o modelo de processo ágil. Uma medida modificada de pontos de função, acoplada com as etapas resumidas na Seção 26.9.1, pode ser usada para desenvolver uma estimativa para a WebApp. Roetzheim [Roe00] sugere a seguinte abordagem ao adaptar pontos de função para estimativa de WebApp:

- *Entradas (Inputs)*. Cada tela de entrada ou formulário [form] (por exemplo, CGI ou Java), cada tela de manutenção e, se você usa uma metáfora de conjunto de guias em algum lugar [tab notebook], cada guia [tab].
- *Saídas (Outputs)*. Cada página da Web estática, cada script de página da Web dinâmica (por exemplo, ASP, ISAPI ou outro script DHTML) e cada relatório (baseado na Web ou de natureza administrativa).
- *Tabelas (Tables)*. Cada tabela lógica na base de dados e, se você estiver usando XML para armazenar dados em um arquivo, cada objeto XML (ou coleção de atributos XML).
- *Interfaces (Interfaces)*. Retêm sua definição como arquivos lógicos (por exemplo, formatos únicos de registro) em nossas fronteiras de saída do sistema.
- *Consultas (Queries)*. Publicadas externamente ou usam uma interface orientada por mensagem. Um exemplo típico são as referências externas DCOM ou COM.

Pontos de função (interpretados da maneira descrita) é um indicador razoável do volume para uma WebApp.

Mendes e seus colegas [Men01] sugerem que o volume de uma WebApp é mais bem determinado coletando-se medidas (chamadas de “variáveis de previsão”) associadas à aplicação (por exemplo, número de páginas, contagem de mídias, número de funções), suas características de páginas da Web (por exemplo, complexidade da página, complexidade de vínculo, complexidade gráfica), características da mídia (por exemplo, duração da mídia) e características funcionais (por exemplo, tamanho do código, tamanho do código reutilizado). Essas medidas podem ser usadas para desenvolver modelos empíricos de estimativa do esforço total de projeto, do esforço para criação de página, do esforço de autorização de mídia e do esforço de scripting. No entanto, há ainda mais trabalho a ser feito para que esses modelos possam ser usados com segurança.

FERRAMENTAS DO SOFTWARE

**Estimativa de mão de obra e custo**

Objetivo: o objetivo das ferramentas de estimativa de esforço e custo é fornecer a uma equipe de projeto as estimativas de esforço necessário, duração do projeto e custo, de uma maneira que trate das características específicas do projeto e o ambiente no qual o projeto deve ser criado.

Mecânica: em geral, as ferramentas de estimativa de custo usam uma base de dados histórica derivada de projetos e dados locais coletados na prática, e um modelo empírico (por exemplo, COCOMO II) utilizado para derivar as estimativas de esforço, duração e custo. As características do projeto e o ambiente de desenvolvimento constituem a entrada, e a ferramenta proporciona como saída uma variedade de estimativas.

Ferramentas representativas:¹⁵

Costar, desenvolvida pela Softstar Systems (www.softstarsystems.com), usa o modelo COCOMO II para desenvolver estimativas de software.

Cost Xpert, desenvolvida pelo Cost Xpert Group, Inc. (www.costxpert.com), integra múltiplos modelos de estimativa e uma base de dados históricos de projeto.

Estimate Professional, desenvolvida pelo Software Productivity Centre, Inc. (www.spc.com), é baseada em COCOMO II e no modelo SLIM.

Knowledge Plan, desenvolvida pela Software Productivity Research (www.spr.com), usa entrada de pontos de função como guia principal para um pacote completo de estimativas.

Price S, desenvolvida pela Price Systems (www.pricesystems.com), é uma das ferramentas de estimativa mais antigas e amplamente utilizadas para projetos de desenvolvimento de software de larga escala.

SEER/SEM, desenvolvida pela Galorath, Inc. (www.galorath.com), fornece recursos de estimativa abrangentes, análise de sensibilidade, avaliação de risco e outras características.

SLIM-Estimate, desenvolvida pela QSM (www.qsm.com), faz uso de “conhecimento industrial” abrangente para proporcionar uma “verificação de plausibilidade” para estimativas derivadas usando dados locais.

26.10 A DECISÃO FAZER/COMPRAR

Em grande número de áreas de aplicação de software, muitas vezes é mais viável em termos de custo adquirir em vez de desenvolver software de computador. Os gerentes de engenharia de software precisam tomar a decisão fazer/comprar, que pode ser ainda mais complicada por uma série de opções de aquisição: (1) o software pode ser comprado (ou licenciado) pronto para uso, (2) componentes de software “totalmente testados” ou “parcialmente testados” (veja a Seção 26.4.2) podem ser adquiridos e depois modificados e integrados para atender a necessidades específicas, ou (3) o software pode ser criado de forma personalizada por um terceiro contratado que atenda às especificações do comprador.

As etapas envolvidas na aquisição de software são definidas pelo aspecto crítico do software a ser comprado e custo final. Em alguns casos (por exemplo, software de PC de baixo custo), é menos dispendioso comprar e experimentar do que fazer uma demorada avaliação dos pacotes de software potenciais. Na análise final, a decisão fazer/comprar é tomada com base nas seguintes condições: (1) A data de entrega do software será mais adiantada comparada com o software desenvolvido internamente? (2) O custo da aquisição mais o custo da personalização serão inferiores ao custo para desenvolver o software internamente? (3) O custo do suporte externo (por exemplo, um contrato de manutenção) será menor do que o custo do suporte interno? Essas condições se aplicam a cada uma das opções de aquisição.

26.10.1 Criando uma árvore de decisões

As etapas que acabamos de descrever podem ser ampliadas por meio de técnicas estatísticas como a árvore de análise de decisões.¹⁶ Por exemplo, a Figura 26.8 mostra uma árvore de decisões para um sistema X baseado em software. Nesse caso, a organização de engenharia de software pode (1) criar o sistema X a partir do zero, (2) reutilizar componentes existentes parcialmente testados para criar o sistema, (3) comprar um software disponível e modificá-lo para atender às necessidades locais, ou (4) contratar o desenvolvimento de software externamente.

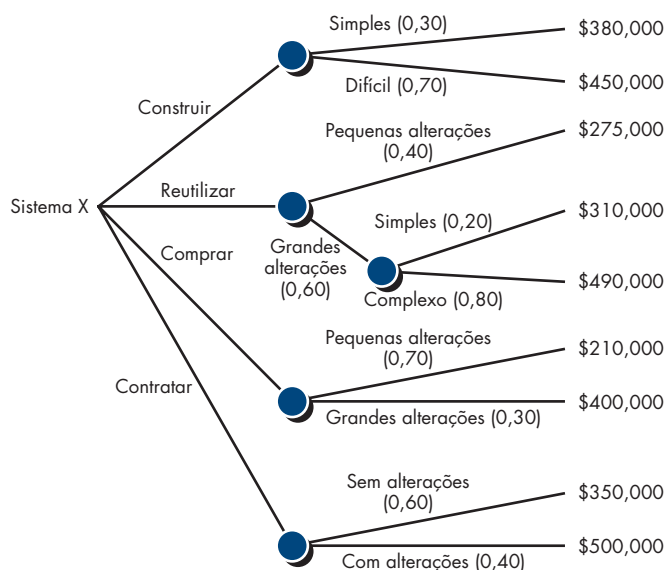
? Existe uma maneira sistemática de escolher entre as opções associadas com a decisão fazer/comprar?

¹⁵ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

¹⁶ Uma boa introdução à árvore de análise de decisões pode ser encontrada no endereço http://en.wikipedia.org/wiki/Decision_tree.

FIGURA 26.8

Uma árvore de decisões para apoiar a decisão fazer/comprar



Se o sistema tem de ser criado desde o início, há uma probabilidade de 70% de que a tarefa será difícil. Usando as técnicas de estimativa discutidas neste capítulo, o planejador do projeto estima que um trabalho de desenvolvimento difícil custará \$ 450 mil. Um trabalho de desenvolvimento “simples” tem um custo estimado de \$ 380 mil. O valor esperado do custo, calculado ao longo de qualquer ramo da árvore de decisões, é

$$\text{Custo esperado} = \sum (\text{probabilidade do caminho})_i \times (\text{custo estimado do caminho})_i$$

em que i é o caminho da árvore de decisões. Para o caminho da criação,

$$\text{Custo esperado}_{\text{construção}} = 0,30 (\$ 380\text{K}) + 0,70 (\$ 450\text{K}) = \$ 429\text{K}$$

Seguindo outros caminhos da árvore de decisões, são mostrados os custos projetados para reutilização, compra e contratação, sob várias circunstâncias. Os custos esperados para esses caminhos são

$$\text{Custo esperado}_{\text{reutilizar}} = 0,40 (\$ 275\text{K}) + 0,60 [0,20 (\$ 310\text{K}) + 0,80 (\$ 490\text{K})] = \$ 382\text{K}$$

$$\text{Custo esperado}_{\text{comprar}} = 0,70 (\$ 210\text{K}) + 0,30 (\$ 400\text{K}) = \$ 267\text{K}$$

$$\text{Custo esperado}_{\text{contratar}} = 0,60 (\$ 350\text{K}) + 0,40 (\$ 500\text{K}) = \$ 410\text{K}$$

Com base na probabilidade e nos custos projetados da Figura 26.8, o custo esperado mais baixo é o da opção “comprar”.

No entanto, é importante observar que muitos outros critérios – não apenas o custo – devem ser considerados durante o processo de tomada de decisões. A disponibilidade, a experiência do desenvolvedor/fornecedor/contratado, a conformidade com os requisitos, as “políticas” locais e a possibilidade de alterações são apenas alguns dos critérios que podem afetar a decisão final de construir, reutilizar, comprar ou contratar.

26.10.2 Terceirização

Mais cedo ou mais tarde, toda empresa que desenvolve software faz a pergunta fundamental: “Há alguma maneira de obtermos o software e sistemas que precisamos a um preço mais baixo?”. A resposta não é nada simples, e as discussões emocionais que ocorrem sempre levam a uma única palavra: *terceirização*.

Conceitualmente, a terceirização é extremamente simples. As atividades de engenharia de software são contratadas de um terceiro que faz o trabalho a um custo menor e, espera-se, com

melhor qualidade. O trabalho de software realizado em uma empresa é reduzido a uma atividade de gerenciamento de contrato.¹⁷

A decisão de optar pela *terceirização* pode ser estratégica ou tática. No nível estratégico, os gerentes de negócio consideram se uma parte significativa de todo o trabalho de software pode ser contratada com outros. No nível tático, um gerente de projeto determina se parte ou todo o projeto pode ser mais bem executado subcontratando o trabalho de software.

Independentemente da amplitude do foco, a decisão de terceirizar muitas vezes é financeira. Uma discussão detalhada da análise financeira para a terceirização está além do escopo deste livro; aconselhamos a consulta de outros autores (por exemplo, [Min95]). No entanto, apresentaremos aqui um rápido exame dos prós e contras dessa decisão.

No lado positivo, podem ser obtidas reduções de custos diminuindo-se o número de pessoas empregadas e instalações (por exemplo, computadores, infraestrutura) que os suportam. No lado negativo, a empresa perde algum controle sobre o software de que precisa. Como software é uma tecnologia que diferencia seus sistemas, serviços e produtos, uma empresa corre o risco de colocar o destino de sua competitividade nas mãos de um terceiro.

A tendência para a terceirização certamente continuará. A única maneira de atenuá-la é reconhecer que o trabalho de software é extremamente competitivo em todos os níveis. A única maneira de sobreviver é se tornar tão competitivo quanto os próprios fornecedores de terceirização.

“Como regra, a terceirização requer uma habilidade de gerenciamento ainda maior do que o desenvolvimento interno.”

Steve McConnell

CASA SEGURA



Terceirização

Cena: Sala de reuniões da CPI Corporation no início do projeto.

Atores: Mal Golden, gerente sênior do setor de desenvolvimento de produto; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo do setor de desenvolvimento de negócio; e Doug Miller, gerente de projeto de engenharia de software.

Conversa:

Joe: Estamos considerando a terceirização da parte de engenharia de software do *CasaSegura*.

Doug (chocado): Quando isso ocorreu?

Lee: Recebemos uma cotação de um desenvolvedor do exterior. Ela está 30% abaixo daquilo que o seu grupo acha que custará. Veja. [Passa a cotação para Doug, que a lê.]

Mal: Como você sabe, Doug, estamos tentando baixar os custos em 30%, e 30% são 30%. Além disso, esse pessoal é bastante recomendado.

Doug (tomando fôlego e tentando manter-se calmo): Bem, vocês me pegaram de surpresa. Antes de tomarmos a decisão final, gostaria de fazer alguns comentários.

Joe (concordando): Claro, prossiga.

Doug: Nunca trabalhamos com essa empresa de terceirização antes, certo?

Mal: Certo, mas...

Doug: E eles afirmam que quaisquer alterações nas especificações serão cobradas a uma taxa adicional, certo?

Joe (contrariado): Certo, mas esperamos que as coisas se mantenham razoavelmente estáveis.

Doug: Uma péssima suposição, Joe.

Joe: Bem...

Doug: É provável que iremos lançar novas versões deste produto durante os próximos anos. E é razoável supor que o software trará muitas das novas características, certo? [Todos acenam afirmativamente.]

Doug: Nós já coordenamos um projeto internacional antes?

Lee (parecendo preocupado): Não, mas me disseram...

Doug (tentando conter sua raiva): Então o que você está me dizendo é que: (1) vamos trabalhar com um fornecedor desconhecido, (2) os custos para fazer isso não são tão baixos quanto parecem, (3) estamos de fato concordando em trabalhar com eles durante muitas versões do produto, não importa o que fizerem na primeira versão, e (4) vamos aprender sobre um projeto internacional “com o bonde andando”. [Todos permanecem em silêncio.]

Doug: Pessoal... Eu acho que isso é um erro, e gostaria que vocês reconsiderassem por mais um dia. Teremos muito mais controle se fizermos o trabalho internamente. Temos a experiência, e posso garantir que não vai nos custar muito mais... O risco será menor e sei que todos vocês têm aversão a riscos, como eu também tenho.

Joe (contrariado): Você apresentou boas razões, mas tem um interesse claro em manter esse projeto internamente.

Doug: Isso é verdade, mas isso não muda os fatos.

Joe (com um suspiro): Ok, vamos adiar por um ou dois dias, pensar um pouco e nos reunimos novamente para uma decisão final. Doug, posso falar em particular com você?

Doug: Claro... Eu realmente quero ter certeza de que estamos fazendo a coisa certa.

17 A terceirização pode ser vista em geral como qualquer atividade que leva à aquisição de software ou componentes de software de uma fonte fora da organização de engenharia de software.

26.11 RESUMO

Um planejador de projeto de software deve estimar três itens antes de começar: quanto tempo levará, quanto esforço será necessário e quantas pessoas serão envolvidas. Além disso, o planejador deve prever os recursos (hardware e software) que serão necessários e o risco envolvido.

A definição de escopo ajuda o planejador a desenvolver estimativas usando uma ou mais técnicas que se dividem em duas grandes categorias: decomposição e modelagem empírica. As técnicas de decomposição requerem um delineamento das principais funções do software, seguido de estimativas de (1) número de LOC, (2) valores selecionados dentro do domínio de informações, (3) número de casos de uso, (4) número de pessoas-mês necessário para implementar cada função, ou (5) número de pessoas-mês necessário para cada atividade de engenharia de software. Técnicas empíricas usam expressões derivadas empiricamente para esforço e tempo para prever esses valores de projeto. Podem ser utilizadas ferramentas automáticas para implementar um modelo empírico específico.

Estimativas precisas de projeto em geral usam pelo menos duas das três técnicas que acabamos de descrever. Comparando e reconciliando as estimativas desenvolvidas por meio de diferentes técnicas, o planejador tem maior possibilidade de derivar uma estimativa precisa. A estimativa de projeto de software nunca pode ser uma ciência exata, mas uma combinação de bons dados históricos e técnicas sistemáticas pode melhorar a precisão da estimativa.

PROBLEMAS E PONTOS A PONDERAR

26.1. Suponha que você seja o gerente de projeto de uma empresa que cria software para robôs de uso doméstico. Você foi contratado para criar o software para um robô que corta a grama de uma residência. Apresente uma definição de escopo que descreva o software. Certifique-se de que a sua definição de escopo esteja delimitada. Se não estiver familiarizado com robôs, faça algumas pesquisas antes de começar a escrever. Além disso, formule suas hipóteses sobre o hardware necessário. Alternativa: Troque o robô cortador de grama por um outro problema de seu interesse.

26.2. A complexidade do projeto de software é discutida rapidamente na Seção 26.1. Desenvolva uma lista de características de software (por exemplo, operação concorrente, saída gráfica) que afetam a complexidade de um projeto. Defina prioridades na lista.

26.3. O desempenho é uma consideração importante durante o planejamento. Discuta como pode ser interpretado diferentemente, dependendo da área de aplicação do software.

26.4. Faça uma decomposição funcional do software do robô que você descreveu no Problema 26.1. Estime o tamanho de cada função em LOC. Supondo que a sua organização produza 450 LOC/pm com um valor bruto de mão de obra de \$ 7 mil por pessoa-mês, estime a mão de obra e custo necessário para criar o software usando a técnica de estimativa baseada em LOC descrita neste capítulo.

26.5. Use o modelo COCOMO II para estimar o esforço necessário para criar software para um caixa eletrônico simples que produz 12 telas, 10 relatórios e requer aproximadamente 80 componentes de software. Suponha que a complexidade seja média e que a maturidade desenvolvedor/ambiente seja média. Use o modelo de composição de aplicação com pontos de objetos.

26.6. Use a equação do software para estimar o software do robô cortador de grama. Assuma que a Equação (26.4) seja aplicável e que $P = 8.000$.

26.7. Compare as estimativas de esforço derivadas nos Problemas 26.4 e 26.6. Qual o desvio-padrão, e como ele afeta o seu grau de certeza sobre a estimativa?

26.8. Com os resultados obtidos no Problema 26.7, determine se é razoável esperar que o software possa ser criado nos próximos seis meses e quantas pessoas serão empregadas para isso.

26.9. Desenvolva um modelo de planilha que implemente uma ou mais das técnicas de estimativa descritas neste capítulo. Como alternativa, use um ou mais modelos on-line para estimativa de fontes da Web.

26.10. Para uma equipe de projeto: desenvolver uma ferramenta de software que implemente cada uma das técnicas de estimativa desenvolvidas neste capítulo.

26.11. Parece estranho que as estimativas de custo e de cronograma sejam feitas durante o planejamento do projeto de software – antes de fazer a análise detalhada dos requisitos de software ou projeto. Por que você acha que isso é feito? Há circunstâncias nas quais não deveria ser feito?

26.12. Calcule novamente os valores esperados para a árvore de decisões da Figura 26.8 supondo que cada ramo tem uma probabilidade 50-50. Como isso muda a sua decisão final?

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Muitos livros de gerenciamento de projeto de software contêm discussões de estimativas de projeto. O Project Management Institute (*PMBOK Guide*, PMI, 2001), Wysoki e seus colegas (*Effective Project Management*, Wiley, 2000), Lewis (*Project Planning Scheduling and Control*, 3d ed., McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3d ed., Wiley, 2000), e Phillips [Phi98] relacionam informações úteis sobre estimativas.

McConnell (*Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006) escreveu um guia prático que fornece amplas diretrizes para qualquer um que precise estimar custo de software. Parthasarathy (*Practical Software Estimation*, Addison-Wesley, 2007) destaca pontos de função como uma métrica de estimativa. Laird e Brennan (*Software Measurement and Estimation: A Practical Approach*, Wiley-IEEE Computer Society Press, 2006) tratam de medida e seu uso na estimativa de software. Pfleeger (*Software Cost Estimation and Sizing Methods, Issues, and Guidelines*, RAND Corporation, 2005) desenvolveu um guia resumido que trata de muitos fundamentos de estimativas. Jones (*Estimating Software Costs*, 2d ed., McGraw-Hill, 2007) escreveu um dos mais abrangentes tratamentos de modelos e dados aplicáveis a estimativas de software em todos os domínios de aplicação. Coombs (*IT Project Estimation*, Cambridge University Press, 2002) e Roetzheim e Beasley (*Software Project Cost and Schedule Estimating: Best Practices*, Prentice-Hall, 1997) apresentam muitos modelos úteis e sugerem diretrizes passo a passo para gerar as melhores estimativas possíveis.

Há disponível na Internet uma ampla variedade de fontes de informação sobre estimativa de software. Pode-se encontrar uma lista atualizada de referências da Web no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CONCEITOS-CHAVE

acompanhamento	.638
caminho crítico	...631
distribuição de	
esforço	634
pessoas e esforço	.632
princípios de	
cronogramas para	
WebApps	641
caixa de tempo	...640
gráfico de Gantt	...638
rede de tarefas	...636
subdivisão do	
trabalho	638
valor agregado	...643

No fim da década de 1960, um jovem e brilhante engenheiro foi selecionado para “escrever” um programa para uma aplicação de fábrica automatizada. O motivo de ter sido escolhido era simples. Ele era a única pessoa em seu grupo que havia participado de um seminário sobre programação de computadores. Conhecia detalhes da linguagem assembly e FORTRAN, mas não sabia nada sobre engenharia de software e menos ainda sobre cronograma e acompanhamento de projeto.

Seu chefe lhe deu os manuais apropriados e uma descrição do que precisava ser feito, além de informá-lo de que o projeto tinha de ficar pronto em dois meses.

Ele leu os manuais, escolheu sua abordagem e começou a escrever o código. Duas semanas depois, o chefe o chamou ao escritório e perguntou como as coisas estavam andando.

“Excelente”, disse o jovem engenheiro com um entusiasmo juvenil. “Era mais simples do que eu pensava. Já estou com quase 75% do trabalho pronto.”

O chefe sorriu e encorajou o jovem engenheiro a continuar seu excelente trabalho. Marcaram outra reunião para uma semana depois.

Uma semana depois, o chefe chamou o engenheiro ao escritório e perguntou, “Como estamos?”.

“Tudo está indo muito bem”, disse o jovem, “mas estou enfrentando alguns obstáculos. Vou resolver tudo isso e voltar logo à rotina normal”.

“Como está o prazo de entrega?”, perguntou o chefe.

“Sem problemas”, disse o engenheiro. “Estou quase com 90% do trabalho pronto”.

PANORAMA

O que é? Você selecionou um modelo de processo apropriado, já identificou as tarefas de engenharia de software que precisam ser executadas, estimou a mão de obra e o número de pessoas, conhece os prazos e até já considerou os riscos. Agora chegou a hora de ligar os pontos. Isto é, tem de criar uma rede de tarefas de engenharia de software que lhe permitirá terminar o trabalho no prazo. Uma vez criada a rede, precisa designar um responsável para cada tarefa, verificar se ela é feita e adaptá-la à medida que os riscos se tornam realidade. Resumindo, isso se chama cronograma e acompanhamento de projeto de software.

Quem realiza? Em nível de projeto, os gerentes de projeto de software usando informações solicitadas dos engenheiros de software. Em nível individual, os próprios engenheiros de software.

Por que é importante? Para criar um sistema complexo, muitas tarefas de engenharia de software ocorrem em paralelo, e o resultado do trabalho executado durante uma tarefa pode ter um profundo efeito sobre o trabalho a ser executado em outra tarefa. Essas interdependências são muito

difíceis de entender sem um cronograma. É também praticamente impossível avaliar o progresso em um projeto de software de tamanho moderado ou grande sem um cronograma detalhado.

Quais são as etapas envolvidas? As tarefas de engenharia de software ditadas pelo modelo de processo de software são refinadas para a funcionalidade a ser criada. Duração e esforço são alocados para cada tarefa e é criada uma rede de tarefas (também chamada de “rede de atividades”) para possibilitar que a equipe de software cumpra os prazos de entrega estabelecidos.

Qual é o artefato? É produzido o cronograma de projeto e informações relacionadas.

Como garantir que o trabalho foi realizado corretamente? Um cronograma apropriado requer que: (1) todos os riscos apareçam na rede, (2) esforço e duração sejam alocados inteligentemente para cada tarefa, (3) as interdependências entre as tarefas sejam indicadas corretamente, (4) sejam alocados recursos para o trabalho a ser feito, e (5) sejam alocados pontos de verificação bem próximos uns dos outros para que o progresso possa ser acompanhado.

Se você já trabalhou no mundo do software por alguns anos, pode imaginar como acaba essa história. Não ficará surpreso em saber que o jovem engenheiro¹ manteve-se nos 90% do trabalho completo durante toda a duração do projeto e terminou (com a ajuda de outros) com um mês de atraso.

Essa história tem se repetido dezenas de milhares de vezes pelos desenvolvedores de software durante as últimas cinco décadas. A grande pergunta é: por quê?

27.1 CONCEITOS BÁSICOS

Embora haja muitas razões para atrasos na entrega do software, muitas delas podem ser atribuídas a uma ou mais das seguintes causas básicas:

- Um prazo de entrega não realístico estabelecido por alguém de fora da equipe de software e imposto sobre os gerentes e profissionais.
- Alterações nos requisitos do cliente não refletidas em alterações no cronograma.
- Uma subestimativa honesta do esforço e/ou quantidade de recursos que serão necessários para executar o serviço.
- Riscos previsíveis e/ou não previsíveis não considerados quando o projeto foi iniciado.
- Dificuldades técnicas que não puderam ser previstas com antecedência.
- Dificuldades humanas que não puderam ser previstas com antecedência.
- Falha de comunicação entre o pessoal de projeto que resulta em atrasos.
- Falha do gerente de projeto em não perceber o atraso do cronograma do projeto e, desse modo, não realizar nenhuma ação para corrigir o problema.

"Cronogramas apertados ou irracionais é provavelmente a influência individual mais destrutiva em todo o software."

Capers Jones

Prazos de entrega agressivos (leia-se "não realísticos") são fato comum nos negócios de software. Às vezes são exigidos por motivos até legítimos, do ponto de vista daquele que define esses prazos. Mas o bom senso indica que a legitimidade deve ser entendida também pelas pessoas que estão executando o trabalho.

Napoleão disse certa vez: "Qualquer comandante-chefe que se propõe a executar um plano que considera falho está cometendo um erro; ele deve apresentar suas razões, insistir para que o plano seja alterado e, por fim, solicitar seu afastamento em vez de ser o instrumento do fracasso de seu exército". Essas são palavras fortes que muitos gerentes de projeto de software deveriam considerar.

As atividades de estimativa discutidas no Capítulo 26 e as técnicas de cronograma discutidas neste capítulo são muitas vezes implementadas sob a pressão de um prazo de entrega definido. Se as melhores estimativas indicam que o prazo de entrega não é realístico, um gerente de projeto competente deverá "proteger sua equipe contra pressões indevidas sobre o cronograma... E enviar a pressão de volta para aqueles que a originaram" [Pag85].

"Adoro prazos de entrega. Gosto do som sibilante que eles fazem quando passam voando."

Douglas Adams

Para ilustrar, suponha que a sua equipe de software tenha sido encarregada de desenvolver um controlador em tempo real para um instrumento de diagnóstico médico que deve ser lançado no mercado em nove meses. Após cuidadosa estimativa e análise de riscos (Capítulo 28), você chega à conclusão de que o software, da maneira como foi solicitado, levará 14 meses corridos para ser criado com o pessoal disponível. Como deve proceder?

Não há sentido ir ao escritório do cliente (nesse caso, o provável cliente é o departamento de marketing/vendas) e pedir que a data de entrega seja alterada. As pressões de marketing externo ditaram a data, e o produto tem de ser entregue. É igualmente uma tolice recusar o trabalho (do ponto de vista profissional). Então, o que fazer? Recomendo tomar as seguintes providências nessa situação:

¹ Caso esteja imaginando, essa história é autobiográfica.

O que devemos fazer quando a gerência nos exige um prazo de entrega impossível?

1. Faça uma estimativa detalhada usando dados históricos de projetos anteriores. Determine o esforço estimado e a duração do projeto.
2. Usando um modelo incremental de processo (Capítulo 2), desenvolva uma estratégia de engenharia de software que fornecerá a funcionalidade crítica no prazo de entrega imposto, mas deixe outras funcionalidades para depois. Documente o plano.
3. Reúna-se com o cliente e (usando a estimativa detalhada), explique por que o prazo de entrega imposto não é praticável. Não deixe de destacar que todas as estimativas são baseadas no desempenho de projetos anteriores. Não deixe também de indicar a porcentagem de melhoria necessária para cumprir o prazo de entrega da forma como ele está definido.² É apropriado fazer o seguinte comentário:

Acredito que temos um problema com o prazo de entrega do software controlador XYZ. Já encaminhei a cada um de vocês um resumo das taxas de produtividade de desenvolvimento de projetos realizados e uma estimativa que já fizemos de muitas maneiras diferentes. Vocês devem observar que considerei uma melhoria de 20% nas taxas de produtividade anteriores, mas ainda temos um prazo de entrega de 14 meses, em vez de 9 meses.

4. Ofereça a estratégia de desenvolvimento incremental como uma alternativa:

Tenho algumas opções a oferecer e gostaria que vocês tomassem a decisão com base nelas. Primeiro, podemos aumentar o orçamento e buscar recursos adicionais para termos certeza de completar o trabalho em nove meses. Mas entendo que isso aumentará o risco de má qualidade devido ao prazo apertado.³ Segundo, podemos remover uma série de funções e recursos do software que vocês estão solicitando. Isso tornará a versão preliminar do produto um pouco menos funcional, mas podemos anunciar uma funcionalidade completa e fornecê-la após 14 meses. Terceiro, podemos ignorar toda a realidade e esperar completar o projeto em nove meses. Vamos acabar não tendo nada a entregar ao cliente. A terceira opção, espero que concordem comigo, é inaceitável. Dados anteriores e nossas melhores estimativas dizem que isso não é realístico: é uma receita para o desastre.

Haverá protestos, mas se for apresentada uma estimativa sólida baseada em bons dados históricos, é provável que sejam escolhidas versões negociadas das opções 1 ou 2. O prazo de entrega não realístico é eliminado.

27.2 CRONOGRAMA DE PROJETO

Perguntaram certa vez a Fred Brooks como é que os projetos de software acabam se atrasando. Sua resposta foi tão simples quanto profunda: “Um dia de cada vez”.

A realidade de um projeto técnico (pode envolver a construção de uma usina hidrelétrica ou o desenvolvimento de um sistema operacional) é que centenas de pequenas tarefas devem ocorrer para se atingir o objetivo maior. Algumas dessas tarefas estão fora da rotina principal e podem ser completadas sem muita preocupação quanto ao impacto na data de entrega do projeto. Outras tarefas estão no “caminho crítico”. Se essas tarefas “críticas” atrasarem-se, o prazo de entrega do projeto inteiro é ameaçado.

Como gerente de projetos, o seu objetivo é definir todas as tarefas, criar uma rede que mostre suas interdependências, identificar as tarefas críticas dentro da rede e acompanhar o progresso para garantir que os atrasos sejam detectados “um dia de cada vez”. Para tanto, deve ter um cronograma definido com um grau de resolução que permita monitorar o progresso e controlar o projeto.

Cronograma de projeto de software é uma atividade que distribui o esforço estimado por toda a duração planejada do projeto alocando esse esforço para tarefas específicas de engenharia de software. No entanto, é importante notar que o cronograma evolui com o tempo. Durante os



As tarefas necessárias para que o gerente de projetos atinja seus objetivos não devem ser executadas manualmente. Há muitas ferramentas excelentes para cronogramas. Use-as.

² Se a melhoria exigida for de 10 a 25%, pode ser possível fazer o trabalho no prazo. Mas é mais provável que a melhoria de desempenho da equipe precisará ser maior do que 50%. Essa é uma expectativa não realística.

³ Você pode argumentar também que o aumento do número de pessoas não reduz proporcionalmente o prazo.

“A sobreposição de cronogramas otimistas não resulta em cronogramas reais mais breves, resulta em mais longos.”

Steve McConnell

PONTO-CHAVE

Ao desenvolver um cronograma, divida o trabalho, anote as dependências entre as tarefas, atribua esforço e tempo para cada tarefa e defina responsabilidades, resultados e pontos de controle.

primeiros estágios do planejamento do projeto, desenvolve-se um cronograma macroscópico. Este identifica as principais atividades do processo e as funções do produto para as quais se aplicam. Conforme o projeto caminha, cada item é refinado em um cronograma detalhado. Nesse momento, ações e tarefas de software específicas (necessárias para realizar uma atividade) são identificadas e dispostas em um cronograma.

O cronograma para projetos de engenharia de software pode ser visto sob duas perspectivas bem diferentes. Na primeira, uma data final para entrega de um sistema computacional já foi definida (de maneira irreversível). A organização de software é forçada a distribuir o esforço no prazo prescrito. A segunda visão considera que os limites cronológicos aproximados foram discutidos, mas a data final é definida pela organização de engenharia de software. O esforço é distribuído para fazer o melhor uso dos recursos, e uma data final é refinada após cuidadosa análise do software. Infelizmente, a primeira situação ocorre com muito maior frequência do que a segunda.

27.2.1 Princípios básicos

Assim como em todas as outras áreas da engenharia de software, há uma série de princípios básicos que guiam os cronogramas de projeto de software:

Divisão do trabalho. O projeto deve ser dividido em uma série de atividades e tarefas gerenciáveis. Para tanto, o produto e o processo são refinados.

Interdependência. Deve ser determinada a interdependência de cada atividade ou tarefa resultante da divisão. Algumas tarefas devem ocorrer em sequência, enquanto outras podem acontecer em paralelo. Certas atividades não podem começar enquanto o resultado de uma outra não esteja disponível. Outras atividades podem ocorrer de forma independente.

Alocação do tempo. Para cada tarefa a ser programada deve ser alocado certo número de unidades de trabalho (por exemplo, pessoas-dias de esforço). Além disso, para cada tarefa deve ser definida uma data de início e uma data de término, que são uma função das interdependências e se o trabalho será realizado em tempo integral ou parcial.

Validação do esforço. Cada projeto tem um número definido de pessoas na equipe de software. Na medida em que ocorre a alocação do tempo, você deve assegurar que, em determinado momento, não seja programado mais do que o número alocado de profissionais. Por exemplo, considere um projeto para o qual são designados três engenheiros de software (três pessoas-dia estão disponíveis por dia de esforço atribuído⁴). Em determinado dia, sete tarefas concorrentes devem ser executadas. Cada tarefa requer 0,50 pessoas-dia de esforço. Foi alocado mais esforço do que pessoas disponíveis para fazer o trabalho.

Definição de responsabilidades. Cada tarefa que é disposta em cronograma deverá ser atribuída a um membro específico da equipe.

Definição dos resultados. Cada tarefa disposta em cronograma deve ter um resultado definido. Para projetos de software, o resultado normalmente é um artefato (por exemplo, o projeto de um componente) ou uma parte de um artefato. Artefatos muitas vezes são combinados em entregáveis.

Definição dos pontos de controle (milestones). Cada tarefa ou grupo de tarefas deve estar associada a um ponto de controle no projeto. Um ponto de controle é atingido quando um ou mais artefatos teve sua qualidade examinada (Capítulo 15) e foi aprovado.

Cada um desses princípios é aplicado à medida que o projeto evolui.

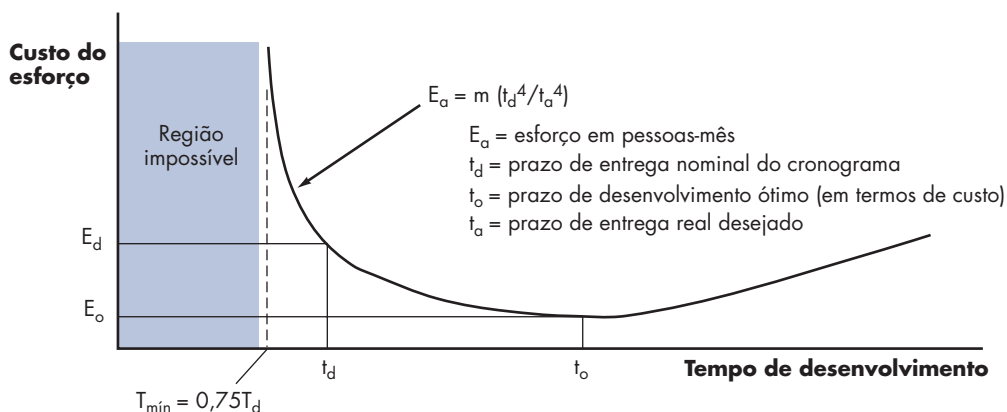
27.2.2 Relação entre pessoas e esforço

Em um pequeno projeto de desenvolvimento de software uma única pessoa pode analisar os requisitos, fazer o projeto, gerar o código e realizar os testes. Conforme o tamanho do projeto aumenta, mais profissionais devem ser envolvidos. (Raramente podemos nos dar ao luxo de considerar o esforço de dez pessoas-ano sendo executado por um só indivíduo trabalhando por dez anos!)

⁴ Na realidade há menos de 3 pessoas-dia disponíveis devido a reuniões, ausência por doença, férias e uma variedade de outras razões. Para nossos propósitos, no entanto, vamos considerar uma disponibilidade de 100%.

FIGURA 27.1

A relação entre esforço e prazo de entrega



Se você tiver de acrescentar pessoas a um projeto atrasado, não se esqueça de atribuir-lhes trabalho que já esteja bastante dividido.

Há um mito comum no qual ainda acreditam muitos gerentes responsáveis por projetos de desenvolvimento de software: “Se nos atrasarmos, podemos sempre acrescentar mais programadores e recuperar o tempo perdido mais tarde”. Infelizmente, acrescentar pessoas nas últimas fases de um projeto muitas vezes tem um efeito prejudicial, fazendo o cronograma se arrastar ainda mais. Os profissionais incluídos precisam aprender sobre o sistema, e os encarregados de ensiná-los são os mesmos que estavam fazendo o trabalho. Enquanto explicam, nada é feito, e o projeto torna-se ainda mais atrasado.

Além do tempo necessário para conhecer o sistema, mais pessoas aumentam o número de caminhos de comunicação e a complexidade das comunicações em todo o projeto. Embora a comunicação seja absolutamente essencial para o desenvolvimento bem-sucedido do software, cada novo caminho requer esforço adicional e, portanto, tempo adicional.

Durante anos, dados empíricos e análises teóricas têm demonstrado que os cronogramas de projeto são elásticos. É possível abreviar uma data de conclusão desejada para um projeto (acrescentando recursos) até certo ponto. É possível também estender uma data de conclusão (reduzindo o número de recursos).

A curva Putnam-Norden-Rayleigh (PNR)⁵ fornece uma indicação da relação entre esforço aplicado e prazo de entrega para um projeto de software. Uma versão da curva, representando esforço de projeto em função do prazo de entrega, é apresentada na Figura 27.1. A curva indica um valor mínimo t_o , que representa o custo mínimo para a entrega (o prazo de entrega que resultará no trabalho mínimo despendido). Quando nos movemos para a esquerda de t_o (quando tentamos acelerar a entrega), a curva sobe não linearmente.

Como exemplo, suponhamos que uma equipe de projeto tenha estimado que um nível de esforço E_d será necessário para conseguir um prazo de entrega nominal t_d que é ótimo em termos de cronograma e recursos disponíveis. Embora seja possível acelerar a entrega, a curva sobe de forma muito aguda para a esquerda de t_d . De fato, a curva PNR indica que o prazo de entrega do projeto não pode ser comprimido além de $0,75t_d$. Se tentarmos uma compressão maior, o projeto passará para a “região impossível”, e o risco de fracasso se torna muito alto. A curva PNR indica também que para a opção de prazo de entrega de custo mais baixo, $t_o = 2t_d$. A implicação aqui é que o atraso na entrega pode reduzir os significativamente custos. Naturalmente, isso deve ser ponderado, levando-se em conta as consequências comerciais associadas com o atraso.

A equação do software [Put92] introduzida no Capítulo 26 é derivada da curva PNR e demonstra a relação altamente não linear entre o tempo cronológico para completar um projeto e o esforço humano aplicado ao projeto. O número de linhas de código produzidas, L , está relacionado a esforço e tempo de desenvolvimento pela equação:

$$L = P \times E^{1/3} t^{4/3}$$

PONTO-CHAVE

Se a entrega pode ser atrasada, a curva PNR indica que os custos do projeto podem ser reduzidos substancialmente.

5 As pesquisas originais podem ser encontradas em [Nor70] e [Put78].

em que E é o esforço de desenvolvimento em pessoas-mês, P é um parâmetro de produtividade que reflete uma variedade de fatores que levam a uma alta qualidade do trabalho de engenharia de software (valores típicos para P variam entre 2.000 e 12.000) e t é a duração do projeto em meses corridos.



Conforme o prazo de entrega se torna cada vez mais apertado, você chega a um ponto em que o trabalho não pode ser terminado no prazo, independentemente do número de pessoas que estejam trabalhando. Enfrente a realidade e defina uma nova data de entrega.

Rearrajando essa equação de software, podemos chegar a uma expressão para trabalho de desenvolvimento E :

$$E = \frac{L^3}{P^3 t^4} \quad (27.1)$$

em que E é o esforço despendido (em pessoas-ano) durante todo o ciclo de vida do desenvolvimento de software e manutenção e t é o tempo de desenvolvimento em anos. A equação para o esforço de desenvolvimento pode ser relacionada com o custo do desenvolvimento pela inclusão de um fator inflacionado da taxa de trabalho (\$/pessoa-ano).

Isso leva a alguns resultados interessantes. Considere um projeto de software complexo, de tempo real, estimado em 33.000 LOC, 12 pessoas-ano de esforço. Se forem atribuídas 8 pessoas para a equipe de projeto, o projeto pode ser feito em aproximadamente 1,3 ano. No entanto, se estendermos a data final para 1,75 ano, a natureza altamente não linear do modelo descrito na Equação (27.1) nos dará:

$$E = \frac{L^3}{P^3 t^4} \sim 3,8 \text{ pessoas-ano}$$

Isso implica que, estendendo em seis meses a data de entrega, podemos reduzir o número de pessoas de oito para quatro! A validade desses resultados está aberta ao debate, mas a consequência é clara: podem ser obtidos benefícios usando menos pessoas durante um período de tempo um pouco mais longo para atingir o mesmo objetivo.

27.2.3 Distribuição de esforço

Cada uma das técnicas de estimativa de projeto de software discutidas no Capítulo 26 leva a estimativas de unidades de trabalho (por exemplo, pessoa-mês) necessárias para completar o desenvolvimento do software. Uma distribuição recomendada do trabalho durante o processo de software muitas vezes é conhecida como *regra 40-20-40*. Quarenta por cento de todo o esforço é alocado na análise preliminar e projeto. Uma porcentagem similar é aplicada ao teste posterior. Você pode inferir corretamente que a codificação (20% do trabalho) está em segundo plano.

Essa distribuição do esforço é usada apenas como guia.⁶ As características de cada projeto ditam a distribuição de esforços. O trabalho despendido em planejamento de projeto raramente passa de 2 a 3% do trabalho, a menos que o plano induza uma organização a grandes despesas com alto risco. A comunicação com o cliente e as análises de requisitos podem chegar de 10 a 25% do trabalho do projeto. O esforço gasto em análise e protótipo deve aumentar em proporção direta com o tamanho e complexidade do projeto. Uma faixa de 20 a 25% do esforço é aplicada normalmente ao projeto de software. Deve ser considerado também o tempo gasto para revisão de projeto e subsequente iteração.

Devido ao esforço aplicado no projeto do software, a codificação deve seguir com relativamente pouca dificuldade. Pode-se aceitar um intervalo de 15 a 20 por cento do esforço total. O teste e o debugging subsequente podem totalizar 30 a 40 por cento do esforço de desenvolvimento do software. A criticidade do software muitas vezes determina o volume de teste necessário. Se o software estiver relacionado com vidas humanas (isto é, a falha do software pode resultar em perda de vidas humanas), as porcentagens podem ser tipicamente mais altas.

? Como o trabalho deve ser distribuído no fluxo do processo de software?

⁶ Hoje, a regra 40-20-40 é criticada. Alguns acreditam que se deveria despendar mais de 40% do esforço total durante a análise e o projeto. Por outro lado, alguns defensores do desenvolvimento ágil (Capítulo 3) argumentam que deveria ser gasto menos tempo "no início" e que uma equipe deveria passar rapidamente para a construção.

27.3 DEFININDO UM CONJUNTO DE TAREFAS PARA O PROJETO DE SOFTWARE

Independentemente do modelo de processo escolhido, o trabalho que uma equipe executa é obtido por meio de uma série de tarefas que permitem definir, desenvolver e, por fim, suportar software de computador. Não há um conjunto único de tarefas que seja apropriado para todos os projetos. O conjunto de tarefas que seria apropriado para um sistema grande e complexo, provavelmente seria considerado exagerado para um software pequeno e razoavelmente simples. Portanto, um processo de software eficaz definiria uma coleção de conjuntos de tarefas, cada uma delas projetada para atender às necessidades de diferentes tipos de projetos.

Conforme mencionamos no Capítulo 2, um conjunto de tarefas é uma coleção de tarefas de engenharia de software, pontos de controle, artefatos e filtros de garantia de qualidade que precisam ser obtidos para completar um projeto em particular. O conjunto de tarefas deve proporcionar disciplina o bastante para obter alta qualidade do software. Mas, ao mesmo tempo, não deve sobrecarregar a equipe com trabalho desnecessário.

Para desenvolver um cronograma de projeto, um conjunto de tarefas deve ser distribuído ao longo da duração do projeto. O conjunto irá variar dependendo do tipo de projeto e do grau de rigor com que a equipe decide fazer seu trabalho. Embora seja difícil desenvolver uma classificação abrangente de tipos de projeto, muitas organizações de software encontram os seguintes projetos:

WebRef

Um modelo de processo adaptável (*Adaptable Process Model* — APM) foi desenvolvido para ajudar na definição de conjuntos de tarefas para vários projetos de software. Uma descrição completa do APM pode ser encontrada no site www.rspa.com/apm.

1. *Projetos de desenvolvimento de conceito* iniciados para explorar algum conceito novo de negócio ou aplicação de uma nova tecnologia.
2. *Projetos de desenvolvimento de novas aplicações* feitos em consequência de uma solicitação de um cliente específico.
3. *Projetos de aperfeiçoamento de aplicação* ocorrem quando um software existente passa por grandes modificações em sua função, desempenho ou interfaces observáveis pelo usuário final.
4. *Projetos de manutenção de aplicação* corrigem, adaptam ou ampliam software existente de maneira não muito óbvia ao usuário final.
5. *Projetos de reengenharia* empreendidos com a intenção de recriar um sistema existente (legado) no todo ou em parte.

Mesmo em um único tipo de projeto, muitos fatores influenciam o conjunto de tarefas a ser selecionado. Esses fatores incluem [Pre05]: tamanho do projeto, número de usuários potenciais, importância da missão, longevidade da aplicação, estabilidade dos requisitos, facilidade de comunicação cliente/desenvolvedor, maturidade da tecnologia aplicável, restrições de desempenho, características internas e não internas, pessoal de projeto e fatores de reengenharia. Quando tomados de forma combinada, esses fatores fornecem uma indicação do *grau de rigor* com que o processo de software deve ser aplicado.

27.3.1 Exemplo de conjunto de tarefas

Projetos de desenvolvimento de conceito ocorrem quando deve ser explorado o potencial para alguma nova tecnologia. Não há certeza de que a tecnologia será aplicável, mas um cliente (por exemplo, marketing) acredita que existem benefícios potenciais. Projetos de desenvolvimento de conceito são abordados aplicando as seguintes ações:

- 1.1 **Definição do escopo do conceito** determina o escopo geral do projeto.
- 1.2 **Planejamento preliminar do conceito** estabelece a capacidade da organização em assumir o trabalho originado pelo escopo do projeto.
- 1.3 **Avaliação do risco da tecnologia** avalia o risco associado à tecnologia a ser implementada como parte do escopo de projeto.
- 1.4 **Prova de conceito** demonstra a viabilidade de uma nova tecnologia no contexto de software.

1.5 Implementação do conceito implementa a representação do conceito de maneira que possa ser examinada por um cliente e usada para finalidades de “marketing” quando um conceito deve ser vendido para outros clientes ou gerentes.

1.6 Reação do cliente ao conceito que solicita informações sobre um novo conceito de tecnologia e focaliza aplicações especiais do cliente.

Um rápido exame dessas ações resultará em algumas surpresas. Na verdade, o fluxo da engenharia de software para projetos de desenvolvimento do conceito (e para todos os outros tipos de projetos também) não é nada mais do que bom senso.

27.3.2 Refinamento das ações de engenharia de software

As ações de engenharia de software descritas na seção anterior podem ser usadas para definir um cronograma macroscópico para um projeto. No entanto, este deve ser refinado para criar um cronograma de projeto detalhado. O refinamento começa tomando cada ação e decompondo-a em uma série de tarefas (com os artefatos correspondentes e seus pontos de controle).

Como exemplo de decomposição de tarefa, considere a Ação 1.1, Escopo do conceito. O refinamento da tarefa pode ser conseguido usando-se o formato de esboço, mas neste livro empregamos a abordagem de uma linguagem de projeto de processo para ilustrar o fluxo das ações de escopo de conceito:

Definição de tarefa: Ação 1.1 Definição do Escopo do Conceito

1.1.1 Identifique as necessidades, benefícios e clientes potenciais;

1.1.2 Defina saída/controlado desejados e eventos de entrada que orientam a aplicação;

Início da Tarefa 1.1.2

1.1.2.1 RT: Reveja a descrição da necessidade⁷

1.1.2.2 Faça uma lista de saídas/entradas visíveis ao cliente

1.1.2.3 RT: Examine saídas/entradas com o cliente e revise conforme necessário;

Fim da tarefa 1.1.2

1.1.3 Defina a funcionalidade/comportamento para cada função principal;

Início da Tarefa 1.1.3

1.1.3.1 RT: Examine os objetos saída e entrada de dados produzidos na tarefa 1.1.2;

1.1.3.2 Crie um modelo de funções/comportamentos;

1.1.3.3 RT: Examine as funções/comportamentos com o cliente e revise conforme necessário;

Fim da tarefa 1.1.3

1.1.4 Isole os elementos da tecnologia a ser implementados em software;

1.1.5 Pesquise a disponibilidade de software existente;

1.1.6 Defina a viabilidade técnica;

1.1.7 Faça uma estimativa rápida do tamanho;

1.1.8 Crie uma definição do escopo;

Fim da definição: Ação 1.1

As tarefas e subtarefas mencionadas no refinamento da linguagem de projeto de processo formam a base de um cronograma detalhado para a ação de escopo de conceito.

27.4 DEFININDO UMA REDE DE TAREFAS

PONTO-CHAVE

A rede de tarefas é um mecanismo útil para mostrar as dependências entre elas e determinar o caminho crítico.

As tarefas e subtarefas individuais têm interdependências baseadas em sua sequência. Além disso, quando há mais de uma pessoa envolvida em um projeto de engenharia de software, é provável que as atividades e tarefas de desenvolvimento sejam executadas em paralelo. Quando isso ocorre, tarefas concorrentes devem ser coordenadas para que estejam prontas quando outras mais adiante necessitarem de seus artefatos.

Uma *rede de tarefas*, também chamada de *rede de atividades*, é uma representação gráfica do fluxo de tarefas de um projeto. Às vezes é usada como um mecanismo por meio do qual a sequência

⁷ RT indica que deve ser feita uma revisão técnica (Capítulo 15).

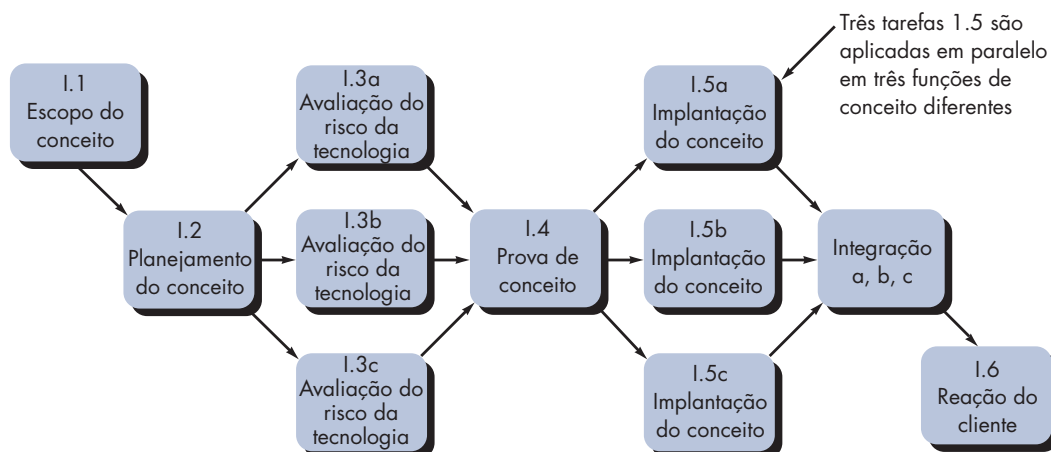
e dependências de tarefa são colocadas em uma ferramenta automática de cronograma de projeto. Em sua forma mais simples (usada ao criar um cronograma macroscópico), a rede de tarefas representa as principais ações da engenharia de software. A Figura 27.2 mostra uma rede de tarefas esquemática para um projeto de desenvolvimento de conceito.

A natureza concorrente das ações de engenharia de software leva a uma série de requisitos importantes para a elaboração de cronograma. Como tarefas paralelas ocorrem de forma assíncrona, você deverá determinar as dependências entre elas para garantir o progresso contínuo até o término do trabalho. Além disso, atenção àquelas tarefas que ficam no *caminho crítico*. Isto é, tarefas que devem ser terminadas no prazo para que o projeto como um todo possa encerrar no prazo. Esses problemas serão discutidos em mais detalhes ainda neste capítulo.

É importante notar que a rede de tarefas da Figura 27.2 é macroscópica. Em uma rede de tarefas detalhada (o precursor de um cronograma detalhado), cada ação na figura seria expandida. Por exemplo, a Tarefa 1.1 seria expandida para mostrar todas as tarefas detalhadas no refinamento das Ações 1.1 mostradas na Seção 27.3.2.

FIGURA 27.2

Uma rede de tarefas para desenvolvimento de conceito



27.5 CRONOGRAMA

“Tudo o que temos de decidir é o que fazer com o tempo que nos é concedido.”

Gandalf, em O senhor dos anéis: a sociedade do anel

O cronograma de um projeto de software não difere muito do cronograma de qualquer esforço de engenharia multitarefa. Portanto, ferramentas e técnicas generalizadas de cronogramas podem ser aplicadas com poucas modificações aos projetos de software.

A *técnica de avaliação e revisão de programa* (*Program Evaluation and Review Technique - PERT*) e o *método do caminho crítico* (*Critical Path Method - CPM*) são dois métodos de cronograma de projetos que podem ser aplicados ao desenvolvimento de software. Ambas as técnicas são controladas por informações já desenvolvidas em atividades anteriores de planejamento de projeto: estimativas de esforço, uma decomposição da função do produto, a escolha do modelo de processo apropriado e o conjunto de tarefas e decomposição das tarefas que estão selecionadas.

As interdependências entre as tarefas podem ser definidas por meio de uma rede de tarefas. Tarefas, também chamadas de *estrutura de subdivisão do trabalho* (*Work Breakdown Structure - WBS*) do projeto, são definidas para o produto como um todo ou para funções individuais.

Tanto PERT quanto CPM fornecem ferramentas quantitativas que lhe permitem (1) determinar o caminho crítico – a cadeia de tarefas que determinam a duração do projeto, (2) estabelecer estimativas de tempo “mais prováveis” para tarefas individuais aplicando modelos estatísticos, e (3) calcular “tempos-limite” que definem uma “janela” de tempo para uma tarefa em particular.

FERRAMENTAS DO SOFTWARE

**Cronograma de projeto**

Objetivo: o objetivo das ferramentas de cronograma de projeto é permitir que o gerente defina as tarefas; estabeleça suas dependências; atribua recursos humanos às tarefas e desenvolva uma variedade de cartas, diagramas e tabelas que ajudem a acompanhar e controlar o projeto de software.

Mecânica: em geral, as ferramentas de cronograma de projeto requerem a especificação de uma estrutura de subdivisão do trabalho das tarefas ou a geração de uma rede de tarefas. Uma vez definido o desmembramento (um esboço) ou rede de tarefas, datas de início e fim, recursos humanos, prazos de entrega e outros dados são anexados a cada uma delas. A ferramenta gera então uma variedade de cartas de tempos e outras tabelas que permitem ao gerente avaliar o fluxo de tarefas. Esses dados podem ser atualizados continuamente no decorrer do projeto.

Ferramentas representativas:⁸

AMS Realtime, desenvolvida pela Advanced Management Systems (www.amsusa.com), tem recursos de cronograma para projetos de todos os tamanhos e tipos.

Microsoft Project, desenvolvida pela Microsoft (www.microsoft.com), é a ferramenta de cronograma de projeto baseada em PC mais amplamente utilizada.

4C, desenvolvida pela 4C Systems (www.4csys.com), suporta todos os aspectos do planejamento de projeto incluindo cronograma.

Uma lista abrangente de fornecedores e artefatos de software de gerenciamento de projeto pode ser encontrada no site www.infogol.com/pmc/pmcswr.htm.

27.5.1 Gráfico de Gantt

Ao criar o cronograma de um projeto de software, você começa com um conjunto de tarefas (a estrutura de subdivisão do trabalho). Se forem usadas ferramentas automáticas, a subdivisão de trabalho entra como uma rede de tarefas ou resumo de tarefas. Dados de esforço, duração e data de início são então definidos para cada tarefa. Além disso, as tarefas podem ser atribuídas a indivíduos específicos.

Como resultado dessas informações, é gerado um *gráfico de Gantt* (*Gantt chart*). Um gráfico de Gantt pode ser desenvolvido para o projeto inteiro, ou podem ser desenvolvidos gráficos separados para cada função do projeto ou para cada indivíduo que trabalha no projeto.

A Figura 27.3 ilustra o formato de um gráfico de Gantt. Ela mostra uma parte de um cronograma de projeto de software que destaca a tarefa de escopo do conceito para um software processador de texto (*Word Processor* – WP). Todas as tarefas do projeto (para escopo do conceito) são listadas na coluna da esquerda. As barras horizontais indicam a duração de cada tarefa. Quando ocorrem múltiplas barras ao mesmo tempo no calendário, é sinal de que há concorrência de tarefas. Os losangos indicam pontos de controle.

Uma vez introduzidas as informações necessárias para a geração de uma carta de tempo, a maioria das ferramentas de cronograma de projeto de software produz *tabelas de projeto* – uma listagem tabular de todas as tarefas de projeto, suas datas de início e fim planejada e atual, e uma variedade de informações relacionadas (Figura 27.4). Usadas em conjunto com a carta de tempo, as tabelas de projeto permitem acompanhar o progresso.

27.5.2 Acompanhando o cronograma

Se tiver sido bem desenvolvido, o cronograma de projeto torna-se um roteiro que define as tarefas e pontos de controle a ser acompanhados e controlados à medida que o projeto avança. O acompanhamento pode ser feito de várias maneiras diferentes:

- Promovendo reuniões periódicas sobre o estado do projeto nas quais cada membro da equipe relata o progresso e os problemas

PONTO-CHAVE

Um gráfico de Gantt permite determinar que tarefas serão executadas em determinado ponto no tempo.

“A regra básica do relatório de status de software pode ser resumida em uma única frase: ‘Sem surpresas’.”

Capers Jones

⁸ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.



A melhor indicação do progresso é a conclusão e revisão bem-sucedida de um artefato de software definido.

- Reunindo-se informalmente com os profissionais para obter sua avaliação subjetiva do progresso até o momento e os problemas previstos
- Usando análise de valor agregado (Seção 27.6) para avaliar o progresso quantitativamente

Na realidade, todas essas técnicas de acompanhamento são empregadas por gerentes de projeto experientes.

O controle é usado por um gerente de projeto de software para administrar os recursos do projeto, enfrentar os problemas e dirigir a equipe. Se tudo estiver bem (isto é, o projeto dentro do prazo e do orçamento, as revisões indicando que há progresso real e os pontos de controle estão sendo alcançados), o controle é fácil. Mas quando ocorrem problemas, você deve exercer o seu controle para conciliar todos os itens o mais rápido possível. Depois que um problema foi diagnosticado, recursos adicionais podem ser focalizados na área problemática: o pessoal pode ser realocado ou o cronograma do projeto redefinido.

Quando enfrentam pressões severas de prazo de entrega, os gerentes de projeto experientes às vezes usam uma técnica de cronograma e controle de projeto chamada de *time-boxing* (caixa de tempo) [Jal04]. A estratégia caixa de tempo reconhece que o produto completo pode não estar pronto no prazo de entrega predefinido. Assim, é escolhido um paradigma de software incremental (Capítulo 2) e gerado um cronograma para cada entrega incremental.

As tarefas associadas a cada incremento são então limitadas em tempo. Isso significa que o cronograma para cada tarefa é ajustado retroativamente a partir da data de entrega para o incremento. Uma “caixa” é traçada ao redor de cada tarefa. Quando uma tarefa chega ao limite de sua caixa de tempo (mais ou menos 10%), o trabalho é interrompido e inicia-se a próxima tarefa.

A reação inicial à abordagem caixa de tempo quase sempre é negativa: “Se o trabalho não está pronto, como podemos prosseguir?”. A resposta está na maneira como o trabalho é feito. Quando se atinge o limite da caixa de tempo, é provável que 90% da tarefa já tenha sido feita.⁹ Os restantes 10%, embora importantes, podem (1) ser adiados até o próximo incremento ou (2) serem concluídos mais tarde se for necessário. Em vez de ficar “preso” em uma tarefa, o projeto prossegue em direção à data de entrega.

PONTO-CHAVE

Quando é atingida a data de término de uma tarefa de tempo limitado, o trabalho daquela tarefa é interrompido e a próxima tarefa começa.

27.5.3 Acompanhando o progresso de um projeto orientado a objeto

Embora um modelo iterativo seja a melhor estrutura para um projeto orientado a objeto, o paralelismo de tarefas torna difícil o acompanhamento do projeto. Você pode ter dificuldades para estabelecer pontos de controle significativos para um projeto orientado a objeto porque uma série de coisas diferentes está acontecendo ao mesmo tempo. Em geral, os seguintes pontos de controle principais podem ser considerados “completos” quando os seguintes critérios forem atingidos.

Ponto de controle técnico: Análise orientada a objeto completa

- Todas as classes e a hierarquia de classes foram definidas e revisadas.
- Os atributos de classe e operações associados a uma classe foram definidos e revisados.
- Os relacionamentos entre classes (Capítulo 6) foram estabelecidos e revisados.
- Um modelo comportamental (Capítulo 7) foi criado e revisado.
- As classes reutilizáveis foram identificadas.

Ponto de controle técnico: projeto orientado a objeto completo

- O conjunto de subsistemas foi definido e revisado.
- Classes foram alocadas a subsistemas e revisadas.
- A alocação de tarefas foi estabelecida e revisada.
- As responsabilidades e colaborações foram identificadas.

⁹ Um cínico pode se recordar do ditado: “Os primeiros 90% do sistema tomam 90% do tempo; os restantes 10% tomam 90% do tempo.”

- Os atributos e operações foram atribuídos e revisados.
- O modelo de comunicação foi criado e revisado.

Ponto de controle técnico: a programação orientada a objeto está completa

- Cada nova classe foi implementada em código por meio do modelo de projeto.
- As classes extraídas (de uma biblioteca de reutilização) foram implementadas.
- Foi criado o protótipo ou incremento.

Ponto de controle técnico: teste orientado a objeto

- Foi examinada a exatidão e totalidade dos modelos de análise e projeto orientado a objeto.
- Foi desenvolvida e revisada a rede responsabilidade-colaboração de classe (Capítulo 6).
- Foram criados os casos de teste e feitos os testes em nível de classe (Capítulo 19) para cada classe.
- Os casos de teste foram criados e o teste de conjunto (Capítulo 19) está completo e as classes integradas.
- Foram finalizados os testes de sistema.

Lembrando que o modelo de processo orientado a objeto é iterativo, cada um desses pontos de controle deve ser revisitado conforme diferentes incrementos são entregues ao cliente.

27.5.4 Cronograma para projetos para WebApp

O *cronograma para projeto para WebApp* distribui o esforço estimado ao longo do tempo planejado (duração) para criar cada incremento de WebApp. Isso é conseguido alocando-se o esforço para tarefas específicas. É importante notar, no entanto, que o cronograma para WebApp geral evolui com o tempo. Durante a primeira iteração, é desenvolvido um cronograma macroscópico. Este identifica todos os incrementos de WebApp e projeta as datas nas quais cada um será entregue. À medida que o desenvolvimento de um incremento progride, o item para o incremento no cronograma macroscópico é refinado em um cronograma detalhado. Nesse momento, são identificadas e agendadas tarefas específicas de desenvolvimento (necessárias para executar uma atividade).

Como exemplo de cronograma macroscópico, considere a WebApp **CasaSeguraGarantida.com**. Recordando discussões anteriores do **CasaSeguraGarantida.com**, podem ser identificados sete incrementos para o componente baseado na Web do projeto:

- Incremento 1: Informações básicas da empresa e do produto
- Incremento 2: Informações detalhadas do produto e downloads
- Incremento 3: Cotações do produto e processamento de pedidos do produto
- Incremento 4: Layout do espaço e projeto do sistema de segurança
- Incremento 5: Informações e ordem dos serviços de monitoramento
- Incremento 6: Controle on-line do equipamento de monitoramento
- Incremento 7: Acesso às informações das contas

A equipe consulta e negocia com os interessados e desenvolve um cronograma *preliminar* de entrega para os sete incrementos. Um gráfico de Gantt para esse cronograma está na Figura 27.5.

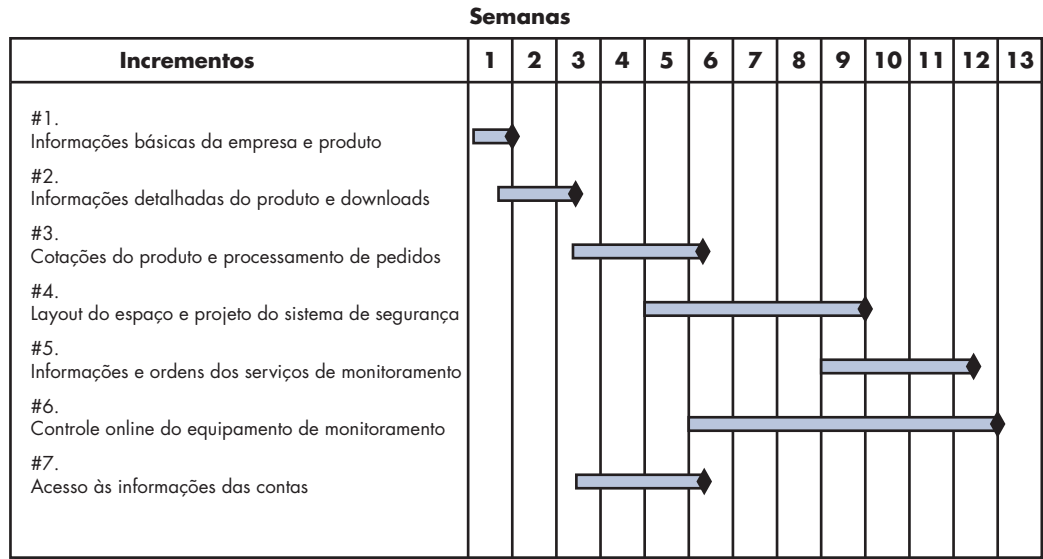
É importante notar que as datas de entrega (representadas por losangos na carta de tempo) são preliminares e podem mudar conforme detalham-se os cronogramas dos incrementos. No entanto, esse cronograma macroscópico fornece ao gerente uma indicação de quando o conteúdo e funcionalidade estarão disponíveis e quando o projeto inteiro estará completo. Como estimativa preliminar, a equipe trabalhará para entregar todos os incrementos em um prazo de 12 semanas. Deve-se notar também que alguns dos incrementos serão desenvolvidos em paralelo (por exemplo, incrementos 3, 4, 6 e 7). Isso implica que a equipe terá pessoas em número suficiente para fazer todo o trabalho em paralelo.



Depuração e teste ocorrem em harmonia. O status da depuração muitas vezes é avaliado considerando-se o tipo e número de erros “pendentes” (bugs).

FIGURA 27.5

Gráfico de Gantt para cronograma macroscópico de projeto



Uma vez desenvolvido o cronograma macroscópico, a equipe está pronta para agendar as tarefas para um incremento específico. Para tanto, pode-se usar a estrutura genérica de processo aplicável a todos os incrementos de WebApp. É criada uma *lista de tarefas* usando as tarefas genéricas derivadas como parte da estrutura como ponto inicial e depois adaptando-as considerando conteúdo e funções a ser derivadas para um incremento de WebApp específico.

Cada ação da estrutura (e suas tarefas relacionadas) pode ser adaptada por uma dentre quatro maneiras: (1) uma tarefa é aplicada como ela está, (2) uma tarefa é eliminada porque não é necessária para o incremento, (3) uma nova tarefa (personalizada) é acrescentada, e (4) uma tarefa é refinada (elaborada) em uma série de subtarefas e cada uma delas se torna parte do cronograma.

Para ilustrar, considere uma ação genérica de *modelagem de projeto* para WebApps que pode ser executada aplicando-se uma ou todas as tarefas a seguir:

- Projeto da estética da WebApp.
- Projeto da interface.
- Projeto do esquema de navegação.
- Projeto da arquitetura da WebApp.
- Projeto do conteúdo e da estrutura que a suporta.
- Projeto dos componentes funcionais.
- Projeto dos mecanismos apropriados de segurança e privacidade.
- Revisão do projeto.

Como exemplo, considere a tarefa genérica *Projeto da Interface* da forma como é aplicada ao quarto incremento de **CasaSeguraGarantida.com**. Lembre-se de que o quarto incremento implementa o conteúdo e função para descrever o espaço residencial ou espaço comercial a ser protegido pelo sistema de segurança *CasaSegura*. De acordo com a Figura 27.5, o quarto incremento começa no início da quinta semana e termina no fim da nona semana.

Não há dúvida de que a tarefa *Projeto da Interface* deve ser executada. A equipe reconhece que o projeto da interface é fundamental para o sucesso do incremento e decide refinar (elaborar) a tarefa. As subtarefas a seguir são derivadas para a tarefa *Projeto da Interface* para o quarto incremento:

- Desenvolver um esboço do layout de página para a página de projeto do espaço.
- Rever o layout com os interessados.
- Projeto dos mecanismos de navegação do layout do espaço.
- Projeto do layout “prancheta de desenho”.¹⁰
- Desenvolver detalhes procedurais para a função de layout da parede gráfica.
- Desenvolver detalhes procedurais para cálculo do comprimento da parede e função de display.
- Desenvolver detalhes procedurais para a função de layout da janela gráfica.
- Desenvolver detalhes procedurais para a função de layout da porta gráfica.
- Projetar mecanismos para selecionar componentes do sistema de segurança (sensores, câmeras, microfones etc.).
- Desenvolver detalhes procedurais para o layout gráfico dos componentes do sistema de segurança.
- Conduzir revisões em pares se necessário.

Essas tarefas se tornam parte do cronograma de incremento para o quarto incremento da WebApp e são alocadas no cronograma de desenvolvimento do incremento. Elas podem ser colocadas no software de cronograma e usadas para acompanhamento e controle.

CASA SEGURA



Acompanhando o cronograma

Cena: Escritório de Doug Miller antes do início do projeto do software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug (observando um slide PowerPoint): O cronograma para o primeiro incremento do *CasaSegura* parece razoável, mas vamos ter problemas para acompanhar o andamento.

Vinod (com cara de preocupado): Por quê? Temos as tarefas dispostas em cronograma diariamente, muitos artefatos, e temos certeza de que não estamos alocando os recursos em excesso.

Doug: Tudo bem, mas como saberemos quando o modelo de requisitos para o primeiro incremento estará completo?

Jamie: As coisas são iterativas, por isso, difíceis.

Doug: Compreendo que, mas... Bem, por exemplo, considere “classes de análise definidas”. Você indicou isso como um ponto de controle.

Vinod: Sim.

Doug: Quem fez essa determinação?

Jamie (sério): Elas estão prontas quando estiverem prontas.

Doug: Isso não é suficiente, Jamie. Temos que agendar RTs [revisões técnicas, Capítulo 15], e você não fez isso. Uma revisão bem-sucedida do modelo de análise, por exemplo, é um ponto de controle razoável. Entendeu?

Jamie (contrariado): Ok, voltemos para a prancheta.

Doug: Não vai levar mais de uma hora para fazer as correções... Todos os demais podem começar já.

27.6 ANÁLISE DE VALOR AGREGADO

PONTO-CHAVE

O valor de retorno proporciona uma indicação quantitativa do progresso.

Na Seção 27.5, discutimos uma série de abordagens quantitativas para o acompanhamento de projeto. Cada uma proporciona ao gerente uma indicação do progresso, mas uma avaliação das informações fornecidas é um tanto subjetiva. É razoável questionar se há uma técnica quantitativa para avaliar o progresso conforme a equipe de software avança nas tarefas alocadas para o cronograma do projeto. Na verdade, existe uma técnica para executar análise quantitativa do progresso. Ela é chamada de *análise de valor agregado* (*Earned Value Analysis* – EVA). Humphrey [Hum95] discute o valor de retorno da seguinte maneira:

¹⁰ Neste estágio, a equipe imagina criar o espaço literalmente desenhando as paredes, janelas e portas por meio das funções gráficas. As linhas das paredes vão se “encaixar” nos pontos de fixação. As dimensões da parede serão mostradas automaticamente. Janelas e portas serão posicionadas graficamente. O usuário pode também selecionar sensores, câmeras etc. específicos e posicioná-los quando o espaço estiver definido.

O sistema de valor agregado proporciona uma escala comum de valor para todas as tarefas [de projeto de software], independentemente do tipo de trabalho que está sendo executado. É estimado o total de horas para completar o projeto, e a cada tarefa é dado um valor agregado com base em sua porcentagem estimada do total.

Em outras palavras, o valor agregado é uma medida do progresso. Permite que você avalie a “porcentagem de conclusão” de um projeto usando análise quantitativa em vez de depender de suposições. De fato, Fleming e Koppleman [Fle98] argumentam que a análise do valor agregado “proporciona leituras precisas e confiáveis do desempenho desde os 15% do projeto”. Para determinar o valor de retorno, executam-se os seguintes passos:

? Como calcular o valor agregado e utilizá-lo para avaliar o progresso?

1. O *custo orçado do trabalho programado* (*Budgeted Cost of Work Scheduled – BCWS*) é determinado para cada tarefa representada no cronograma. Durante a estimativa, é planejado o trabalho (em pessoas-horas ou pessoas-dias) para cada tarefa de engenharia de software. Desse modo, o $BCWS_i$ é o trabalho planejado para a tarefa i . Para determinar o progresso em um dado ponto ao longo do cronograma de projeto, o valor de BCWS é a soma dos valores de $BCWS_i$ para todas as tarefas que deveriam ter sido completadas naquele ponto no tempo no cronograma do projeto.
2. Os valores de BCWS para todas as tarefas são somados para derivar o *orçamento no final* (*Budget At Completion – BAC*). Assim,

$$BAC = \sum (BCWS_k) \text{ para todas as tarefas } k$$
3. Em seguida, é computado o valor para *custo orçado do trabalho executado* (*Budgeted Cost of Work Performed – BCWP*). O valor de BCWP é a soma dos valores de BCWS para todas as tarefas que foram realmente completadas em certo momento no cronograma de projeto.

Wilkens [Wil99] observa que “a distinção entre BCWS e BCWP é que o primeiro representa o orçamento das atividades planejadas para ser completadas, e o último representa o orçamento das atividades que realmente foram completadas”. Ao darmos valores para BCWS, BAC e BCWP, podemos computar importantes indicadores de progresso:

$$\begin{aligned} \text{Índice de desempenho do cronograma (schedule performance index), SPI} &= \frac{BCWP}{BCWS} \\ \text{Variância do cronograma (schedule variance), SV} &= BCWP - BCWS \end{aligned}$$

O SPI é uma indicação da eficiência com a qual o projeto está utilizando os recursos programados. Um valor SPI próximo de 1,0 indica execução eficiente do cronograma de projeto. O SV é apenas uma indicação absoluta da variância em relação ao cronograma planejado.

$$\text{Porcentagem programada para conclusão} = \frac{BCWS}{BAC}$$

fornece uma indicação da porcentagem do trabalho que deveria ter sido completada no tempo t .

$$\text{Porcentagem completada} = \frac{BCWP}{BAC}$$

fornece uma indicação quantitativa da porcentagem completada de um projeto em dado instante t .

É possível também calcular o *custo real do trabalho executado* (*Actual Cost of Work Performed – ACWP*). O valor de ACWP é a soma do trabalho realmente dispendido em tarefas completadas até determinado instante no cronograma de projeto. É possível então calcular

$$\text{Índice de desempenho de custo (cost performance index), CPI} = \frac{BCWP}{ACWP}$$

$$\text{Variância do custo (cost variance), CV} = BCWP - ACWP$$

Um valor de CPI próximo de 1,0 fornece uma forte indicação de que o projeto está de acordo com o seu orçamento. CV é uma indicação absoluta de economia de custos (em relação aos custos planejados) ou déficit em determinado estágio de um projeto.

WebRef

Uma grande variedade de recursos de análise de valor agregado pode ser encontrada no site www.acq.osd.mil/pm/.

Assim como o radar que vigia o horizonte, a análise de valor agregado esclarece as dificuldades do cronograma antes que possam se tornar aparentes. Isso lhe permite tomar as ações corretivas antes que se desenvolva uma crise no projeto.

27.7 RESUMO

O cronograma é o resultado da atividade de planejamento que é um componente primário do gerenciamento de projeto de software. Quando combinado com métodos de estimativa e análise de riscos, o cronograma estabelece um mapa para o gerente de projeto.

O cronograma começa com a decomposição do processo. As características do projeto são empregadas para adaptar um conjunto de tarefas apropriado para o trabalho a ser feito. Uma rede de tarefas mostra cada tarefa de engenharia, sua dependência de outras tarefas e duração projetada. A rede de tarefas é utilizada para calcular o caminho crítico, uma carta de tempo e uma variedade de informações de projeto. Usando o cronograma como guia, você pode acompanhar e controlar cada etapa no processo de software.

PROBLEMAS E PONTOS A PONDERAR

27.1. Prazos de entrega “não razoáveis” são um fato real no negócio de software. Como você procederá se tiver de enfrentar uma situação dessas?

27.2. Qual a diferença entre cronograma macroscópico e cronograma detalhado? É possível gerenciar um projeto se houver apenas o cronograma macroscópico? Por quê?

27.3. Pode haver um caso em que o ponto de controle de um projeto de software não esteja vinculado a uma revisão? Em caso afirmativo, forneça um ou mais exemplos.

27.4. “Sobrecarga de comunicação” pode ocorrer quando múltiplas pessoas trabalham em um projeto de software. O tempo gasto em comunicação reduz a produtividade individual (LOC/mês), e o resultado pode ser menor produtividade para a equipe. Ilustre (quantitativamente) como os engenheiros que são bem versados em boas práticas de engenharia de software e usam revisões técnicas podem aumentar a taxa de produção de uma equipe (quando comparado com a soma das taxas de produção individuais). Dica: Você pode considerar que as revisões reduzem o retrabalho e que o retrabalho pode ser responsável por 20 a 40% do tempo de um profissional.

27.5. Embora acrescentar pessoas a um projeto de software em atraso possa retardá-lo ainda mais, há circunstâncias em que isso não é verdade. Descreva-as.

27.6. A relação entre pessoas e tempo é altamente não linear. Usando a equação do software de Putnam (descrita na Seção 27.2.2), desenvolva uma tabela que relaciona número de pessoas com duração de projeto para um projeto de software que requer 50.000 LOC e 15 pessoas-ano de esforço (o parâmetro produtividade é 5.000 e $B = 0,37$). Considere que o software deve ser entregue em mais de 24 meses ou menos de 12 meses.

27.7. Suponha que você foi contratado por uma universidade para desenvolver um sistema de registro de curso on-line (*online course registration system* - OLCRS). Primeiro, aja como o cliente (se você é um estudante, isso é fácil!) e especifique as características de um bom sistema. (Como alternativa, o seu professor lhe fornecerá uma série de requisitos preliminares para o sistema.) Usando os métodos de estimativa discutidos no Capítulo 26, desenvolva uma estimativa de esforço e duração para OLCRS. Sugira como você faria para:

- Definir atividades paralelas durante o projeto OLCRS.
- Distribuir o esforço através do projeto.
- Estabelecer pontos de controle para o projeto.

27.8. Selecione uma série de tarefas apropriadas para o projeto OLCRS.

27.9. Defina uma rede de tarefas para OLCRS descrita no Problema 27.7 ou, como alternativa, para outro projeto de software que lhe interesse. Não deixe de mostrar as tarefas e os pontos de controle e faça estimativas de trabalho e duração para cada tarefa. Se possível, utilize uma ferramenta de cronograma automática para executar esse trabalho.

27.10. Se há uma ferramenta de cronograma automática, determine o caminho crítico para a rede definida no Problema 27.9.

27.11. Usando uma ferramenta de cronograma (se estiver disponível) ou papel e lápis (se necessário), desenvolva uma carta de tempo para o projeto OLCRS.

27.12. Suponha que você seja o gerente de projeto de software e lhe pediram para computar estatísticas de valor agregado para um pequeno projeto de software. O projeto tem 56 tarefas planejadas que, segundo as estimativas, requerem 582 pessoas-dia para se completar. No instante em que lhe pediram para fazer a análise de valor agregado, 12 tarefas já estavam concluídas. No entanto, o cronograma do projeto indica que 15 tarefas já terão sido completadas. Estão disponíveis os seguintes dados de cronograma (em pessoas-dia):

Tarefa	Esforço planejado	Esforço real
1	12,0	12,5
2	15,0	11,0
3	13,0	17,0
4	8,0	9,5
5	9,5	9,0
6	18,0	19,0
7	10,0	10,0
8	4,0	4,5
9	12,0	10,0
10	6,0	6,5
11	5,0	4,0
12	14,0	14,5
13	16,0	—
14	6,0	—
15	8,0	—

Calcule SPI, variância de cronograma, porcentagem programada para completar, porcentagem completa, CPI e variância de custo para o projeto.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Quase todos os livros escritos sobre gerenciamento de projeto de software contêm uma discussão sobre cronograma. Wysoki (*Effective Project Management*, Wiley, 2006), Lewis (*Project Planning Scheduling and Control*, 4th ed., McGraw-Hill, 2006), Luckey e Phillips (*Software Project Management for Dummies*, For Dummies, 2006), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, 9th ed., Wiley, 2005), Hughes (*Software Project Management*, McGraw-Hill, 2005), The Project Management Institute (*PMBOK Guide*, 3d ed., PMI, 2004), Lewin (*Better Software Project Management*, Wiley, 2001), e Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3d ed., Wiley, 2000) apresentam discussões úteis sobre o assunto. Embora específico de aplicação, Harris (*Planning and Scheduling Using Microsoft Office Project 2007*, Eastwood Harris Pty Ltd., 2007) fornece uma discussão útil sobre como as ferramentas de cronograma podem ser usadas para acompanhar e controlar de forma bem-sucedida um projeto de software.

Fleming e Koppelman (*Earned Value Project Management*, 3d ed., Project Management Institute Publications, 2006), Budd (*A Practical Guide to Earned Value Project Management*, Management Concepts, 2005), e Webb e Wake (*Using Earned Value: A Project Manager's Guide*, Ashgate Publishing, 2003) discutem o uso das técnicas de valor agregado para planejamento de projeto, acompanhamento e controle com considerável detalhe.

Uma ampla variedade de fontes de informação sobre cronograma de projeto de software está disponível na Internet. Uma lista atualizada das referências da Web relevantes para cronogramas de projetos de software pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

**CONCEITOS-
-CHAVE**

análise de	
imprevistos.	658
avaliação	651
categorias de	
risco.	649
componentes e fatores	
de risco	651
estratégias.	649
proativo	649
reativo	649
exposição ao	
risco.	655
identificação	650
previsão.	652
refinamento	656
RMMM.	658
tabela de risco ...	653

Em seu livro sobre análise e gestão de riscos, Robert Charette [Cha89] apresenta uma definição conceitual de risco:

Primeiro, o risco refere-se a acontecimentos futuros. Hoje e ontem já não constituem mais uma preocupação, já que estamos colhendo os resultados de nossas ações. A pergunta é, mudando nossas ações hoje, podemos, então, criar uma oportunidade para uma situação diferente e, conforme esperamos, melhor para nós mesmos amanhã? Segundo, isso significa que o risco envolve mudanças, como de opinião, ações ou lugares... [Terceiro,] o risco envolve escolha e a incerteza que a própria escolha traz. Paradoxalmente, o risco, assim como a morte e os impostos, é uma das poucas certezas da vida.

Quando se considera risco no contexto da engenharia de software, os três fundamentos conceituais de Charette estão sempre em evidência. O futuro é uma preocupação — quais riscos podem fazer o projeto de software dar errado? A alteração é uma preocupação — como as alterações nos requisitos do cliente, nas tecnologias de desenvolvimento, nos ambientes-alvo e todas as outras entidades conectadas ao projeto afetam a cadência e o sucesso geral? Por último, deve-se levar a sério as escolhas — que métodos e ferramentas deve-se usar, quantas pessoas deverão ser envolvidas, quanta ênfase na qualidade seria “suficiente”?

Peter Drucker [Dru75] disse certa vez, “Embora seja tolice querer eliminar o risco, e é questionável tentar minimizá-lo, é essencial que os riscos assumidos sejam os certos”. Para que você possa identificar os “riscos certos” a ser assumidos durante um projeto de software, é importante identificar todos os riscos óbvios tanto para os gerentes quanto para os profissionais.

PANORAMA

O que é? Análise e gestão de risco são ações que ajudam uma equipe de software a entender e gerenciar a incerteza. Muitos problemas podem perturbar um projeto de software. O risco é um problema potencial — ele pode ocorrer ou não. Independentemente do resultado, é aconselhável identificá-lo, avaliar sua probabilidade de ocorrência, estimar seu impacto e estabelecer um plano de contingência caso o problema realmente ocorra.

Quem realiza? Todos aqueles envolvidos na gestão de qualidade — gerentes, engenheiros de software e outros interessados — participam da análise e gestão do risco.

Por que é importante? Pense naquele princípio dos escoteiros: “Sempre alerta”. Software é uma empreitada difícil. Muitas coisas podem dar errado, e, francamente, muitas vezes dão. Por essa razão, estar preparado — entender os riscos e tomar medidas proativas para evitá-los ou administrá-los — é um elemento-chave do bom gerenciamento de projeto de software.

Quais são as etapas envolvidas? Reconhecer o que pode dar errado é o primeiro passo, chamado de “identificação do risco”. Em seguida, o risco é analisado para determinar a probabilidade de que ocorra e o dano que causará, se ocorrer. Uma vez estabelecidas essas informações, os riscos são classificados, por probabilidade e por impacto. Por fim, é desenvolvido um plano para gerenciar os riscos de alta probabilidade e alto impacto.

Qual é o artefato? É produzido um plano de mitigação, monitoramento e gestão de risco (*risk mitigation, monitoring, and management* — RMMM) ou um conjunto de formulários de informações sobre o risco.

Como garantir que o trabalho foi realizado corretamente? Os riscos analisados e gerenciados serão resultantes de um estudo completo das pessoas, produto, processo e projeto. O RMMM deverá ser sempre examinado na medida em que o projeto avança para garantir que os riscos se mantenham atualizados. Os planos de contingência para gerenciamento de risco deverão ser realistas.

28.1 ESTRATÉGIAS DE RISCOS REATIVA VERSUS PROATIVA

"Se você não atacar os riscos ativamente, eles ativamente atacarão você."

Tom Gilb

Estratégias de risco *reativas* têm sido chamadas de forma pejorativa como "Escola Indiana Jones de gestão de risco" [Tho92]. Nos filmes que levam seu nome, Indiana Jones, ao enfrentar uma enorme dificuldade, invariavelmente diria, "Não se preocupe, vou pensar em alguma coisa!". Nunca se preocupe com os problemas até eles acontecerem, Indy reagiria de alguma forma heroica.

Infelizmente, o gerente de projeto de software não é um Indiana Jones, e os membros de sua equipe de projeto de software não são seus fiéis seguidores. No entanto, a maioria das equipes de software depende apenas de estratégias de risco reativas. No melhor dos casos, uma estratégia reativa monitora o projeto à procura de riscos prováveis. São reservados recursos para enfrentar os riscos, caso se tornem problemas reais. Normalmente, a equipe de software não faz nada sobre os riscos até que alguma coisa dê errado. Desse modo, a equipe corre na tentativa de corrigir o problema rapidamente. Isso costuma ser chamado de *modo de combate ao incêndio*. Quando falha, os "gestores de crises" [Cha92] assumem, e o projeto está realmente ameaçado.

Uma estratégia consideravelmente mais inteligente para o gerenciamento de risco é ser proativo. Uma estratégia *proativa* inicia muito antes que o trabalho técnico comece. São identificados os riscos potenciais, avalia-se a probabilidade e o impacto, e os riscos são classificados por ordem de importância. Então, a equipe de software estabelece um plano para gerenciar o risco. O objetivo primário é evitar o risco, mas como nem todos os riscos podem ser evitados, o grupo trabalha para desenvolver um plano de contingência que lhe permita responder de maneira controlada e eficaz. Durante todo o restante deste capítulo, discutiremos uma estratégia proativa de gestão de risco.

28.2 RISCOS DE SOFTWARE

Embora já se tenha debatido muito sobre a definição apropriada para risco de software, há um consenso geral de que o risco sempre envolve duas características: *incerteza* — o risco pode ou não ocorrer, isto é, não existem riscos com probabilidade de 100%¹ — e *perda* — se o risco se tornar uma realidade, ocorrerão consequências indesejadas ou perdas [Hig95]. Quando os riscos são analisados, é importante quantificar o nível de incerteza e o grau de perda associada a cada risco. Para tanto, consideram-se diferentes categorias de riscos.

? Que tipos de riscos você provavelmente encontrará ao criar software?

Riscos de projeto ameaçam o plano do projeto. Isto é, se os riscos do projeto se tornarem reais, é possível que o cronograma fique atrasado e os custos aumentem. Os riscos de projeto identificam problemas potenciais de orçamento, cronograma, pessoal (equipes e organização), recursos, clientes, e requisitos e seu impacto sobre o projeto de software. No Capítulo 26, a complexidade do projeto, tamanho e grau de incerteza estrutural também foram definidos como fatores de risco de projeto (e de estimativa).

Riscos técnicos ameaçam a qualidade e a data de entrega do software a ser produzido. Se um risco técnico potencial se torna realidade, a implementação pode se tornar difícil ou impossível. Os riscos técnicos identificam problemas potenciais de projeto, implementação, interface, verificação e manutenção. Além disso, a ambiguidade de especificações, a incerteza técnica, a obsolescência técnica e a tecnologia "de ponta" também são fatores de risco. Riscos técnicos ocorrem porque o problema é mais difícil de resolver do que se pensava.

Riscos de negócio ameaçam a viabilidade do software a ser criado e muitas vezes ameaçam o projeto ou o produto. Os candidatos aos cinco principais riscos de negócio são (1) criar um excelente produto ou sistema que ninguém realmente quer (risco de mercado), (2) criar um produto que não se encaixe mais na estratégia geral de negócios da empresa (risco estratégico), (3) criar um produto que a equipe de vendas não sabe como vender (risco de vendas), (4) perda de suporte da alta gerência devido à mudança no foco ou mudança de profissionais (risco gerencial), e (5) perda do orçamento ou do comprometimento dos profissionais (riscos de orçamento).

¹ Um risco 100% provável é uma restrição no projeto de software.

"Projetos sem nenhum risco real são fracassos. Eles são quase sempre desprovidos de benefícios; e é por essa razão que não foram feitos anos atrás."

Tom DeMarco e
Tim Lister

É extremamente importante observar que uma simples classificação de risco nem sempre funcionará. Alguns riscos são impossíveis de prever.

Uma outra classificação geral dos riscos foi proposta por Charette [Cha89]. *Riscos conhecidos* são aqueles que podem ser descobertos após uma cuidadosa avaliação do plano do projeto, do ambiente comercial e técnico no qual o projeto está sendo desenvolvido e de outras fontes de informação confiáveis (por exemplo, data de entrega irreal, falta de documentação dos requisitos ou do escopo do software, ambiente de desenvolvimento ruim). *Riscos previsíveis* são extrapolados da experiência de projetos anteriores (por exemplo, rotatividade do pessoal, comunicação deficiente com o cliente, diluição do esforço da equipe conforme as solicitações de manutenção vão sendo atendidas). *Erros imprevisíveis* são o curinga no baralho. Eles podem ou não ocorrer, mas são extremamente difíceis de identificar com antecedência.



Sete princípios da gestão de risco

O Instituto de Engenharia de Software (Software Engineering Institute — SEI) (www.sei.cmu.edu) identifica sete princípios que "proporcionam uma estrutura para conseguir uma gestão de risco eficaz". São eles:

Mantenha uma perspectiva global — encare os riscos de software sob o contexto de um sistema no qual ele é um componente e o problema de negócio que se pretende resolver.

Tenha uma visão antecipada — pense sobre os riscos que podem surgir no futuro (por exemplo, devido a mudanças no software); estabeleça planos de contingência para que os eventos futuros sejam controláveis.

Estimule a comunicação aberta — se alguém apontar um risco potencial, não menospreze. Se um risco é proposto de uma maneira informal, considere-o. Estimule todos os interessados e usuários a sugerir riscos em qualquer instante.

Integre — uma consideração do risco deve ser integrada na gestão de qualidade.

Enfatize um processo contínuo — a equipe deve estar vigilante por toda a gestão de qualidade, modificando os riscos identificados na medida em que mais informações forem conhecidas e acrescentando novos quando uma visão melhor é obtida.

Desenvolva uma visão compartilhada do produto — se todos os interessados compartilham da mesma visão do software, é provável que se tenha uma melhor identificação e avaliação do risco.

Estimule o trabalho de equipe — os talentos, habilidades e conhecimento de todos os interessados deverão ser examinados quando se executam atividades de gestão de risco.

INFORMAÇÕES

28.3 IDENTIFICAÇÃO DO RISCO

A identificação do risco é uma tentativa sistemática para especificar ameaças ao plano do projeto (estimativas, cronograma, recursos etc.). Identificando os riscos conhecidos e previsíveis, o gerente de projeto dá o primeiro passo no sentido de evitá-los quando possível e controlá-los quando necessário.

Há dois tipos distintos de riscos para cada uma das categorias apresentadas na Seção 28.2: riscos genéricos e riscos específicos de produto. *Riscos genéricos* é uma ameaça potencial a todo projeto de software. *Riscos específicos de produto* podem ser identificados somente por aqueles que têm uma visão clara da tecnologia, das pessoas e do ambiente específico para o qual o software está sendo desenvolvido. Para identificar riscos específicos de produto, são examinados o plano do projeto e a definição de escopo do projeto, e procura-se uma resposta para a seguinte pergunta: "Que características especiais desse produto podem ameaçar o plano do nosso projeto?"

Um método para identificar riscos é criar uma *checklist* (lista de verificação) dos itens de risco. Ela pode ser usada para identificação do risco e concentra-se em alguns dos subconjuntos dos riscos conhecidos e previsíveis nas seguintes subcategorias genéricas:

- *Tamanho do produto* — riscos associados ao tamanho geral do software a ser criado ou modificado.
- *Impacto de negócio* — riscos associados a restrições impostas pela gerência ou pelo mercado.



Embora seja importante considerar os riscos genéricos, são os riscos específicos do produto que causam os maiores problemas. Dedique tempo suficiente para identificar tantos riscos de produto quanto for possível.

- *Características do cliente* — são riscos associados à sofisticação dos clientes e à habilidade do desenvolvedor em se comunicar com os interessados a tempo.
- *Definição do processo* — riscos associados ao grau em que a gestão de qualidade foi definida e é seguida pela organização de desenvolvimento.
- *Ambiente de desenvolvimento* — riscos associados à disponibilidade e qualidade das ferramentas a ser usadas para criar o produto.
- *Tecnologia a ser criada* — riscos associados à complexidade do sistema a ser criado e com a “novidade” da tecnologia que está embutida no sistema.
- *Quantidade de pessoas e experiência* — riscos associados à experiência técnica em geral e de projeto dos engenheiros de software que farão o trabalho.

A lista dos itens de risco pode ser organizada de diversas maneiras. Questões relevantes a cada um dos tópicos podem ser respondidas para cada projeto de software. As respostas a essas questões permitem estimar o impacto do risco. Um outro formato de lista de itens de risco simplesmente lista as características relevantes a cada subcategoria genérica. Por fim, é listado um conjunto de “componentes e fatores de risco” [AFC88] com sua probabilidade de ocorrência. Fatores de desempenho, suporte, custo e cronograma são discutidos em resposta às últimas questões.

Há disponível na Web muitas listas abrangentes para riscos de projeto de software (por exemplo, [Baa07], [NAS07], [Wor04]). Você pode usá-las para ter uma visão dos riscos genéricos para projetos de software.

28.3.1 Avaliando o risco geral do projeto

As seguintes questões foram derivadas dos dados de risco obtidos entrevistando gerentes de projeto de software experientes em diversas partes do mundo [Kei98]. As questões estão ordenadas por sua importância relativa ao sucesso de um projeto.

? O projeto de software em que estamos trabalhando está em sério risco?

1. A alta gerência e o cliente estão formalmente comprometidos em apoiar o projeto?
2. Os usuários finais estão bastante comprometidos com o projeto e sistema/produto a ser criado?
3. Os requisitos estão amplamente entendidos pela equipe de engenharia de software e seus clientes?
4. Os clientes foram envolvidos totalmente na definição dos requisitos?
5. Os usuários finais têm expectativas realísticas?
6. O escopo do projeto é estável?
7. A equipe de engenharia de software tem a combinação de aptidões adequadas?
8. Os requisitos de projeto são estáveis?
9. A equipe de projeto tem experiência com a tecnologia a ser implementada?
10. O número de pessoas na equipe de projeto é adequado para o trabalho?
11. Todos os clientes e usuários concordam com a importância do projeto e requisitos para o sistema/produto a ser criado?

WebRef

Risk radar (radar de risco) é uma base de dados e ferramentas que ajudam os gerentes a identificar, classificar e comunicar riscos do projeto. Ela pode ser encontrada no site www.spmn.com.

Se alguma dessas questões for respondida negativamente, devem ser providenciados, imediatamente, processos de mitigação, monitoração e gerenciamento. O grau de risco do projeto é diretamente proporcional ao número de respostas negativas a essas questões.

28.3.2 Componentes e fatores de risco

A Força Aérea Americana [AFC88] publicou um panfleto contendo excelentes diretrizes para identificação e combate a riscos de software. A abordagem da Força Aérea requer que o gerente de projeto identifique os fatores de risco que afetam os componentes de risco de software —

desempenho, custo, suporte e cronograma. No contexto dessa discussão, os componentes de risco são definidos da seguinte maneira:

“Gerenciamento de risco é gerenciamento de projeto para adultos.”

Tim Lister

- *Risco de desempenho* — é o grau de incerteza de que o produto atenderá aos seus requisitos e será adequado para o uso que se pretende.
- *Risco de custo* — é o grau de incerteza de que o orçamento do projeto será mantido.
- *Risco de suporte* — é o grau de incerteza de que o software resultante será fácil de corrigir, adaptar e melhorar.
- *Risco de cronograma* — é o grau de incerteza de que o cronograma do projeto será mantido e que o produto será entregue a tempo.

O impacto de cada motivador de risco sobre o componente de risco é dividido em uma dentre quatro categorias de impacto-negligenciável, marginal, crítico ou catastrófico. Na Figura 28.1 [Boe89], descreve-se uma caracterização das consequências potenciais dos erros (linhas com o título 1) ou falha em obter um resultado desejado (linhas com o título 2).

A categoria de impacto é escolhida com base na caracterização que melhor se adapta à descrição na tabela.

28.4 PREVISÃO DE RISCO

A *previsão de risco*, também chamada de *estimativa de risco*, tenta classificar cada risco de duas maneiras — (1) a possibilidade ou probabilidade de que o risco seja real e (2) as consequências

FIGURA 28.1

Avaliação de impacto
Fonte: (Boe89)

Componentes		Desempenho	Suporte	Custo	Cronograma
Categoria					
Catastrófico	1	Falha em satisfazer o requisito resultaria em falha da missão		A falha resulta em aumento de custo e atrasos no cronograma com valores previstos que excedem \$ 500 mil	
	2	Degradação significativa até não cumprimento do desempenho técnico	Software que não responde com agilidade ou que é difícil de dar suporte	Dificuldades financeiras significativas, provável estouro no orçamento	Data de entrega não exequível
Crítico	1	Falha em atender o requisito degradará o desempenho do sistema até um ponto no qual o sucesso da missão é questionável		Falha resulta em atrasos operacionais e/ou aumento de custos com valores estimados entre \$ 100 mil e \$ 500mil	
	2	Alguma redução no desempenho técnico	Pequenos atrasos nas modificações de software	Alguma falta de recursos financeiros, possíveis estouros de orçamento	Possível atraso na data de entrega
Marginal	1	Falha em atender o requisito resultaria na degradação de missão secundária		Custos, impactos e/ou atrasos de cronograma recuperáveis com valores estimados de \$ 1 mil a \$ 100 mil	
	2	De mínima a pequena redução no desempenho técnico	Suporte responsivo de software	Recursos financeiros suficientes	Cronograma realístico e possível
Negligenciável	1	Falha em atingir o requisito criaria inconveniência ou impacto não operacional		Erro resulta em pequeno impacto no custo e/ou cronograma com valor esperado de menos de \$ 1 mil	
	2	Nenhuma redução do desempenho técnico	Software facilmente suportável	Possível sobra no orçamento	Data de entrega pode ser antecipada

Notas: (1) Potencial consequência de erros ou falhas de software não detectadas.

(2) Potencial consequência se o resultado desejado não é obtido.

dos problemas associados ao risco, caso ele ocorra. Você trabalha com outros gerentes e pessoal técnico para executar quatro etapas de projeção de risco:

1. Estabeleça uma escala que reflita a possibilidade detectada de um risco.
2. Esboce as consequências do risco.
3. Estime o impacto do risco sobre o projeto e o produto.
4. Avalie a exatidão geral da projeção de risco para que não haja mal-entendidos.



Pense seriamente sobre o software que você vai criar e pergunte a si mesmo, “o que pode sair errado?”. Crie sua lista e peça a outros membros da equipe que façam o mesmo.

A finalidade dessas etapas é considerar os riscos de uma maneira que leve à definição de prioridades. Nenhuma equipe de software tem os recursos para resolver todos os riscos possíveis com o mesmo grau de rigor. Priorizando os riscos, você pode alocar recursos onde eles terão maior impacto.

28.4.1 Desenvolvendo uma tabela de risco

Uma tabela de riscos lhe fornece uma técnica simples para a projeção de risco.² Um exemplo é apresentado na Figura 28.2.

Inicia-se listando todos os riscos (não importa quão remotos sejam) na primeira coluna da tabela. Isso pode ser conseguido com a ajuda da lista de itens de risco mencionada na Seção 28.3. Cada risco é caracterizado na segunda coluna (por exemplo, PS implica risco de tamanho de projeto, BU implica risco de negócio). A probabilidade de cada risco é colocada na próxima coluna da tabela. O valor da probabilidade para cada risco pode ser estimado pelos membros da equipe individualmente. Para tanto, pode-se consultar os membros da equipe em ordem aleatória até que suas avaliações coletivas de risco comecem a convergir.

FIGURA 28.2

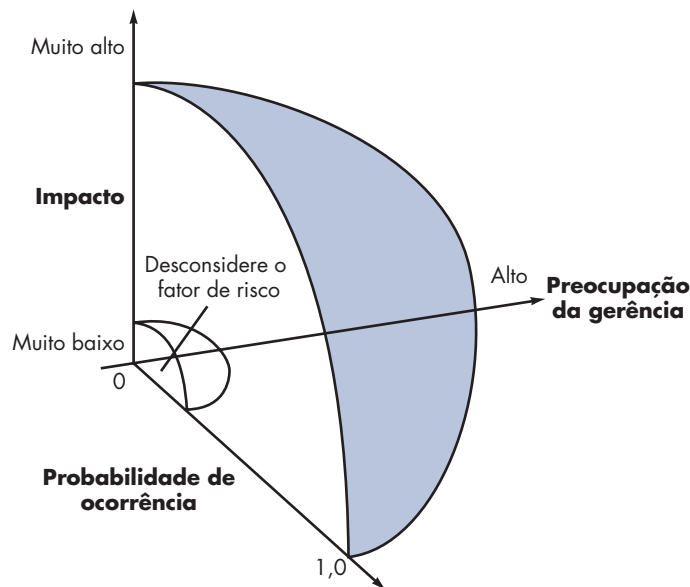
Exemplo de uma tabela de risco antes da ordenação

Riscos	Categoria	Probabilidade	Impacto	RMMM
A estimativa de tamanho pode ser significativamente baixa	PS	60%	2	
Número de usuários maior do que o planejado	PS	30%	3	
Reutilização menor do que a planejada	PS	70%	2	
Os usuários finais resistem ao sistema	BU	40%	3	
O prazo de entrega será apertado	BU	50%	2	
Financiamento será perdido	CU	40%	1	
O cliente mudará os requisitos	PS	80%	2	
A tecnologia não atingirá as expectativas	TE	30%	1	
Falta de treinamento no uso das ferramentas	DE	80%	3	
Pessoal sem experiência	ST	30%	2	
A rotatividade do pessoal será alta	ST	60%	2	
Σ				
Σ				
Σ				

Valores de impacto:

- 1 — catastrófico
- 2 — crítico
- 3 — marginal
- 4 — negligenciável

² A tabela de risco pode ser implementada como um modelo de planilha. Isso permite fácil manipulação e ordenação dos valores.

FIGURA 28.3**Riscos e preocupação da gerência**

Em seguida, avalia-se o impacto de cada risco. É investigado cada componente de risco usando a caracterização apresentada na Figura 28.1, e determina-se uma categoria de impacto. São tomadas as médias das categorias de cada um dos quatro componentes de risco-desempenho, suporte, custo e cronograma³ para determinar um valor de impacto global.

Uma vez completadas as quatro primeiras colunas da tabela, ela é ordenada por probabilidade e por impacto. Riscos de alta probabilidade, alto impacto se situam no topo da tabela, e riscos de baixa probabilidade posicionam-se no fim. Com isso se completa a priorização de risco de primeira ordem.

Você pode estudar a tabela resultante e definir uma linha de corte (traçada horizontalmente em algum ponto na tabela), que implica que somente riscos que ficam acima da linha receberão mais atenção. Riscos que se posicionam abaixo da linha são reavaliados para uma priorização de segunda ordem. De acordo com a Figura 28.3, o impacto e a probabilidade do risco têm influência distinta na preocupação do gerente. Um fator de risco com alto impacto, mas uma probabilidade de ocorrência muito baixa, não deve absorver tempo significativo da gerência. No entanto, riscos de alto impacto com probabilidade entre moderada e alta e riscos de baixo impacto com alta probabilidade devem ser encaminhados para as etapas de análise de risco a seguir.

Todos os riscos que ficam acima da linha de corte deverão ser gerenciados. A coluna com o título RMMM contém um ponteiro que aponta para um plano de *mitigação, monitoramento, e gestão do risco* ou, como alternativa, uma coleção de formulários de informações desenvolvida para todos os riscos que se posicionam acima da linha de corte. O plano RMMM e os formulários de informações de risco são discutidos nas Seções 28.5 e 28.6.

A probabilidade do risco pode ser determinada por meio de estimativas individuais e depois pelo desenvolvimento de um valor de consenso. Apesar de essa abordagem funcionar, foram desenvolvidas técnicas mais sofisticadas para determinar a probabilidade do risco [AFC88]. Os fatores de risco podem ser avaliados em uma escala qualitativa de probabilidades que tem os seguintes valores: impossível, improvável, provável e frequente. A probabilidade matemática pode ser associada a cada valor qualitativo (por exemplo, uma probabilidade de 0,7 a 0,99 envolve um risco altamente provável).

PONTO-CHAVE

A tabela de riscos é ordenada por probabilidade e impacto para classificar os riscos.

"[Hoje] ninguém se dá ao luxo de conhecer uma tarefa tão bem que não possa ter surpresas, e surpresa significa risco."

Stephen Grey

3 Pode ser usada uma média ponderada se um componente de risco tiver mais significado para um projeto.

28.4.2 Avaliando o impacto do risco

Três fatores afetam as consequências prováveis se ocorrer um risco: sua natureza, seu escopo e sua época. A natureza do risco indica os problemas que podem surgir se ele ocorrer. Por exemplo, uma interface externa para o hardware do cliente mal definida (um risco técnico) logo atrapalhará o início do projeto e os testes, e provavelmente causará problemas na integração do sistema no fim do projeto. O escopo de um risco relaciona a severidade (quão sério é ele?) com sua distribuição geral (quanto do projeto será afetado ou quantos clientes serão prejudicados?). Por fim, a época do risco considera quando e por quanto tempo o impacto será sentido. Em muitos casos, você vai querer que as “más notícias” ocorram o mais cedo possível, mas em alguns, quanto mais tarde, melhor.



Retornando mais uma vez à abordagem de análise de risco proposta pela Força Aérea norte-americana [AFC88], podem-se aplicar os seguintes procedimentos para determinar as consequências gerais de um risco: (1) determinar o valor médio da probabilidade de ocorrência para cada componente de risco; (2) usando a Figura 28.1, determinar o impacto para cada componente com base no critério mostrado, e (3) completar a tabela de risco e analisar os resultados conforme descrito nas seções anteriores.

A exposição geral ao risco (*risk exposure* — RE) é determinada por meio da seguinte relação [Hal98]:

$$RE = P \times C$$

em que P é a probabilidade de ocorrência de um risco, e C o custo para o projeto, caso o risco ocorra.

Por exemplo, suponha que a equipe de software defina o risco de um projeto da seguinte maneira:

Identificação do risco. Somente 70% dos componentes de software programados para ser reutilizados serão, de fato, integrados na aplicação. A funcionalidade restante terá de ser desenvolvida de forma personalizada.

Probabilidade do risco. 80% (aproximadamente).

Impacto do risco. Foram planejados 60 componentes de software reutilizáveis. Se somente 70% pode ser usado, 18 componentes terão de ser desenvolvidos desde o início (além de outros softwares personalizados que foram planejados para ser desenvolvidos). Pelo fato de cada componente ter em média 100 LOC e os dados locais indicarem que o custo de engenharia de software para cada LOC é de \$ 14, o custo total (impacto) para desenvolver os componentes será

$$18 \times 100 \times 14 = \$ 25.200.$$

Exposição ao risco. $RE = 0,80 \times 25.200 = \$ 20.200.$



Compare a RE, para todos os riscos, com a estimativa de custos para o projeto. Se RE for maior do que 50% do custo do projeto, a viabilidade do projeto deve ser avaliada.

A exposição ao risco pode ser calculada para cada risco na tabela de riscos, uma vez feita a estimativa do custo do risco. A exposição total para todos os riscos (acima da linha de corte na tabela de riscos) pode proporcionar um meio para ajustar a estimativa final de custo para um projeto. Ela pode também ser usada para prever o aumento provável nos recursos de pessoal necessários em vários pontos durante o cronograma do projeto.

As técnicas de projeção e análise de risco descritas nas Seções 28.4.1 e 28.4.2 são aplicadas iterativamente à medida que avança o projeto de software. A equipe deve rever a tabela de risco a intervalos regulares, reavaliando cada risco para determinar quando novas circunstâncias causam mudanças em probabilidade e impacto. Como uma consequência dessa atividade, pode ser necessário acrescentar novos riscos à tabela, remover alguns que não são mais relevantes e mudar as posições relativas dos que restarem.

CASA SEGURA



Análise de risco

Cena: Escritório de Doug Miller antes do início do projeto de software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe de engenharia.

Conversa:

Doug: Gostaria de dedicar um tempo para um brainstorming sobre os riscos do projeto *CasaSegura*.

Jamie: E o que pode dar errado?

Doug: Bem. Aqui estão algumas categorias em que as coisas podem dar errado. [Ele mostra a todos as categorias listadas na introdução da Seção 28.3.]

Vinod: Humm... Você quer apenas chamar a nossa atenção para eles, ou...

Doug: Não, veja o que acho que devemos fazer. Cada um faz uma lista de riscos... Agora mesmo..." [Dez minutos para todos escreverem.]

Doug: Ok, parem.

Jamie: Mas eu ainda não terminei!

Doug: Tudo bem. Veremos a lista novamente. Agora, para cada item da sua lista, atribua uma porcentagem de probabilidade de

que o risco venha a ocorrer. Depois, atribua um impacto ao projeto em uma escala de 1 (pequeno) a 5 (catastrófico).

Vinod: Se achar que o risco é alto, especifique uma probabilidade de 50%, e se achar que ele tem um impacto moderado sobre o projeto, especifique um 3, certo?

Doug: Exatamente. [Cinco minutos, todos escrevendo.]

Doug: Ok, parem. Agora vamos fazer uma lista de grupos no quadro branco. Eu escrevo; vou pegar um item por vez de cada lista de vocês em uma sequência de rodadas. [Quinze minutos depois, a lista está criada.]

Jamie (apontando para o quadro e rindo): Vinod, aquele risco (apontando para um item do quadro) é ridículo. É mais fácil sermos atingidos por um raio. Deveríamos removê-lo.

Doug: Não, vamos deixar por enquanto. Consideraremos todos os riscos, não importa que sejam absurdos. Depois, vamos limpar a lista.

Jamie: Mas já temos mais de 40 riscos... Como poderemos controlar todos eles?

Doug: Não podemos. É por esse motivo que definiremos um ponto de corte após ordenarmos todos os riscos. Farei isso depois, e nos reunimos amanhã novamente. Por ora, voltemos ao trabalho... E, nos intervalos de folga, pensem sobre quaisquer riscos que tenham esquecido.

28.5 REFINAMENTO DO RISCO

Durante os primeiros estágios do planejamento de projeto, um risco pode ser especificado de maneira bem generalizada. Conforme o tempo passa e se conhece mais sobre o projeto e o risco, pode ser possível refinar o risco em uma série de riscos detalhados. Cada um deles de certa forma mais fácil de mitigar, monitorar e gerenciar.

Uma maneira de fazer isso é representar o risco em um formato *condição-transição-consequência* (CTC) [Glu94]. O risco é definido da seguinte maneira:

Considerando que <condição> então há a preocupação de que (possivelmente) <consequência>.

Usando o formato CTC para o risco de reutilização descrito na Seção 28.4.2, podemos escrever:

Considerando que todos os componentes de software reutilizáveis devem estar em conformidade com padrões específicos de projeto e que alguns deles não se enquadram nesses padrões, há uma preocupação de que (possivelmente) somente 70% dos módulos que se planejava reutilizar possam realmente ser integrados na montagem do sistema, resultando na necessidade de criar de forma personalizada os 30% restantes dos componentes.

Essa condição geral pode ser refinada da seguinte maneira:

Subcondição 1. Certos componentes reutilizáveis foram desenvolvidos por uma equipe terceirizada que não conhecia os padrões internos de projeto.

Subcondição 2. O padrão de projeto para as interfaces de componente não foi completamente estabelecido e pode não estar em conformidade com certos componentes reutilizáveis existentes.

Subcondição 3. Certos componentes reutilizáveis foram implementados em uma linguagem não suportada no ambiente em que serão usados.

? Qual é uma boa maneira de descrever um risco?

As consequências associadas com essas subcondições refinadas permanecem as mesmas (30% dos componentes de software devem ser criados de forma personalizada), mas o refinamento ajuda a isolar os riscos subjacentes e pode levar a uma análise e resposta mais fáceis.

28.6 MITIGAÇÃO, MONITORAÇÃO E CONTROLE DE RISCOS (RMMM)

“Se tomo tantas medidas de precaução, é porque não quero dar chance ao azar.”

Napoleão

Todas as atividades de análise de risco apresentadas até este ponto têm um único objetivo: ajudar a equipe de projeto no desenvolvimento de uma estratégia para lidar com o risco. Uma estratégia eficaz deve considerar três aspectos: como evitar o risco, como monitorar o risco e como gerenciar o risco e planejar contingência.

Se a equipe de software adota uma abordagem proativa ao risco, evitar o risco é sempre a melhor estratégia. Para tanto, desenvolve-se um plano para *mitigação de risco*. Por exemplo, suponha que a alta rotatividade de pessoal seja o risco r_1 de um projeto. Com base em histórico passado e com a intuição do gerente, a possibilidade l_1 de alta rotatividade seja estimada em 0,70 (70%, um tanto alta) e o impacto x_1 seja projetado como crítico. A alta rotatividade terá um impacto crítico sobre o custo e cronograma do projeto.

Para mitigar esse risco, devemos desenvolver uma estratégia para redução da rotatividade. Entre as providências possíveis a ser tomadas, citamos:

- Reunir-se com o pessoal para determinar as causas da rotatividade (por exemplo, más condições de trabalho, salário baixo, mercado de trabalho competitivo).
- Mitigar as causas que estão sob o seu controle antes do início do projeto.
- Uma vez iniciado o projeto, assumir que a rotatividade acontecerá e desenvolver técnicas para garantir a continuidade quando as pessoas saírem.
- Organizar equipes de projeto para que as informações sobre cada atividade de desenvolvimento sejam amplamente difundidas.
- Definir padrões para os produtos do projeto e estabelecer mecanismos para assegurar que todos os modelos e documentos sejam desenvolvidos a tempo.
- Executar revisões em pares de todo o trabalho (para que mais de uma pessoa esteja “por dentro”).
- Designar uma pessoa substituta para cada profissional cujo trabalho seja crítico.

À medida que o projeto avança, começam as atividades de *monitoramento de risco*. O gerente de projeto monitora fatores que podem fornecer uma indicação sobre se o risco está se tornando mais ou menos possível. No caso da alta rotatividade do pessoal, a atitude geral dos membros da equipe baseada nas pressões do projeto, o grau segundo o qual a equipe se tornou coesa, as relações pessoais entre os membros da equipe, problemas em potencial com remuneração e benefícios, e a disponibilidade de empregos dentro e fora da empresa, tudo isso é monitorado.

Além de monitorar esses fatores, um gerente de projeto deve monitorar a efetividade das providências para a mitigação do risco. Por exemplo, uma providência citada recomendava a definição de padrões para o produto e mecanismos para assegurar que os produtos sejam desenvolvidos a tempo. Esse é um mecanismo para garantir a continuidade, se um elemento crítico deixar o projeto. O gerente de projeto deve monitorar os produtos cuidadosamente para garantir que cada um seja autossuficiente e forneça as informações necessárias se um novato precisar entrar na equipe de software no meio do projeto.

O *gerenciamento de risco e plano de contingência* considera que os esforços de mitigação do risco falharam e que o risco se tornou uma realidade. Continuando o exemplo, o projeto está em andamento e um grupo de pessoas avisa que vai sair. Se a estratégia de mitigação foi utilizada, existe pessoal substituto disponível, as informações estão documentadas e todo o conhecimento compartilhado dentro da equipe. Além disso, pode-se temporariamente mudar o foco dos

 O que podemos fazer para mitigar um risco?

recursos (e reajustar o cronograma do projeto) para as funções que estão com o todo o pessoal necessário, permitindo que os novatos a ser acrescentados à equipe “entrem logo no ritmo”. As pessoas que estão saindo devem interromper todo o trabalho e passar suas últimas semanas envolvidas em atividades de “transferência de conhecimentos”. Isso pode incluir captação de conhecimento por meio de vídeo, desenvolvimento de “documentos de comentário ou Wikis”, e/ou reuniões com outros membros da equipe que permanecerão no projeto.



Se a exposição a um risco específico for menor do que seu custo de mitigação, não tente mitigar o risco, mas continue monitorando-o.

É importante observar que as etapas de mitigação, monitoramento e controle de risco (*risk mitigation, monitoring, and management* — RMMM) incorrem em custo adicional no projeto. Por exemplo, o tempo gasto para incluir um substituto para cada técnico essencial custa dinheiro. Parte do gerenciamento de risco é para avaliar quando os benefícios acumulados pelas providências RMMM são superados pelos custos associados a sua implementação. Essencialmente, executa-se uma análise de custo-benefício clássica. Se as providências para evitar os riscos de alta rotatividade aumentarem o custo e duração do projeto em uma estimativa de 15%, mas o fator de custo predominante for o “backup do profissional”, o gerente pode decidir não implementar essa etapa. Por outro lado, se houver uma projeção de que as providências para evitar o risco aumentarão os custos em 5% e a duração em apenas 3%, o gerente provavelmente vai tomar essas providências.

Para um grande projeto, 30 ou 40 riscos podem ser identificados. Se para cada um deles forem identificados de 3 a 7 passos de gestão de risco, esta pode se tornar um projeto em si mesma! Por essa razão, deve-se adaptar a regra 80-20 de Pareto para o risco de software. A experiência indica que 80% do risco geral de projeto (80% do potencial de falha do projeto) pode ser responsável por apenas 20% dos riscos identificados. O trabalho executado durante as primeiras etapas de análise de risco o ajudará a determinar quais dos riscos estão incluídos nesses 20% (por exemplo, riscos que levam à maior alta exposição ao risco). Desse modo, alguns dos riscos identificados, avaliados e projetados podem não entrar no plano RMMM — eles não estão incluídos nos 20% críticos (os riscos com prioridade mais alta no projeto).

O risco não está limitado ao próprio projeto de software. Riscos podem ocorrer depois que o software foi desenvolvido com sucesso e entregue ao cliente. Esses riscos estão tipicamente associados às consequências da falha no software em campo.

Segurança do software e análise de imprevistos (por exemplo, [Dun02], [Her00], [Lev95]) são atividades de garantia de qualidade de software (Capítulo 16) que se concentram na identificação e avaliação de imprevistos em potencial que podem afetar negativamente o software e fazer o sistema inteiro falhar. Se imprevistos puderem ser identificados antecipadamente no processo de engenharia de software, características do projeto do software que servirão para eliminar ou controlar os imprevistos em potencial poderão se especificadas.

28.7 O PLANO RMMM

Uma estratégia de gestão de risco pode ser incluída no plano de projeto de software, ou as etapas de gestão de risco podem ser organizadas em um plano de mitigação, monitoração e gerenciamento (RMMM) separado. O plano RMMM documenta todo o trabalho executado como parte da análise de risco e é usado pelo gerente de projeto como parte do plano geral de projeto.

Algumas equipes de software não desenvolvem um documento RMMM formal. Em vez disso, cada risco é documentado individualmente usando-se um *formulário de informações de risco* (*risk information sheet* — RIS) [Wil97]. Em muitos casos, o RIS é mantido por meio de um sistema de base de dados para que a criação e introdução de informações, ordem de prioridade, pesquisas e outras análises possam ser feitas facilmente. O formato do RIS está ilustrado na Figura 28.4.

Uma vez documentado o RMMM e começado o projeto, iniciam-se as etapas de mitigação e monitoração de risco. Conforme já discutimos, mitigação de risco é uma atividade para evitar problemas. A monitoração de risco é uma atividade de acompanhamento de projeto com três objetivos primários: (1) avaliar se os riscos previstos vão, de fato, ocorrer; (2) assegurar que as

FIGURA 28.4

Formulário de informações de risco

Fonte: [Wil97]

Formulário de informações de risco			
ID do risco: P02-4-32	Data: 09/05/09	Prob: 80%	Impacto: alto
Descrição: Somente 70% dos componentes de software programados para reutilização serão, de fato, integrados na aplicação. A funcionalidade restante terá de ser desenvolvida de maneira personalizada.			
Refinamento/contexto: Subcondição 1: certos componentes reutilizáveis foram desenvolvidos por uma equipe terceirizada que não tinha conhecimento dos padrões de projeto internos. Subcondição 2: o padrão de design para interfaces de componente ainda não foi consolidado e pode não estar em conformidade com certos componentes reutilizáveis. Subcondição 3: certos componentes reutilizáveis foram implementados em uma linguagem não suportada no ambiente a que se destina.			
Mitigação/monitoração: 1. Contate a empresa terceirizada para determinar a conformidade com os padrões de projeto. 2. Pressione para que haja padronização da interface; considere a estrutura de componente ao decidir sobre o protocolo de interface. 3. Determine o número de componentes que estão na categoria da subcondição 3; determine se pode ser adquirido o suporte de linguagem.			
Gerenciamento/plano de contingência/disparo: Foi calculada a exposição ao risco e resultou em \$ 20.200. Reserve esse valor no custo de contingência do projeto. Desenvolva um cronograma revisado assumindo que 18 componentes adicionais terão de ser criados de forma personalizada; defina a equipe de maneira correspondente. Disparo: as providências para mitigação improdutivas em 01/07/09.			
Estado atual: 12/05/09: iniciadas as etapas de mitigação.			
Autor: D. Gagne		Autorizado: B. Laster	

etapas de mitigação ao risco definidas para o risco estejam sendo aplicadas adequadamente; e (3) coletar informações que podem ser usadas para futuras análises de riscos. Em muitos casos, os problemas que ocorrem durante um projeto podem estar ligados a mais de um risco. Outra função do monitoramento de risco é tentar definir a origem [quais riscos causaram quais problemas durante o projeto].

FERRAMENTAS DO SOFTWARE**Gestão da qualidade de software**

Objetivo: o objetivo das ferramentas de gestão de risco é ajudar a equipe de projeto na definição dos riscos, avaliação de seu impacto e probabilidade, e acompanhamento dos riscos durante todo o projeto de software.

Mecânica: em geral, as ferramentas de gestão de risco ajudam na identificação genérica dos riscos fornecendo uma lista de riscos típicos de projeto e comerciais, proporcionando checklists ou outras técnicas de “entrevista” que ajudam a identificar riscos específicos de projeto, atribuindo probabilidade e impacto a cada risco, suportando estratégias de mitigação de risco e gerando vários relatórios diferentes relacionados aos riscos.

Ferramentas representativas:⁴

@risk, desenvolvida pela Palisade Corporation (www.palisade.com), é uma ferramenta genérica de análise de risco que usa simulação de Monte Carlo para controlar seu instrumento analítico.

Riskman, distribuída pela ABS Consulting (www.absconsulting.com/riskmansoftware/index.html), é um sistema especializado de avaliação de risco que identifica riscos relacionados com projeto.

Risk Radar, desenvolvida pela SPMN (www.spmn.com), auxilia os gerentes de projeto na identificação e gerenciamento de riscos de projeto.

Risk+, desenvolvida pela Delttek (www.delttek.com), integra-se com o Microsoft Project para quantificar incerteza de custo e cronograma.

X:PRIMER, desenvolvida pela GrafP Technologies (www.grafp.com) é uma ferramenta genérica baseada na Web que prevê o que pode sair errado em um projeto e identifica as principais causas para falhas em potencial e as ações eficazes a tomar.

⁴ As ferramentas aqui apresentadas não significam um aval, mas, sim, uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

28.8 RESUMO

Sempre que um projeto de software estiver em execução, o bom senso manda que se faça a análise de risco. No entanto, a maioria dos gerentes de projeto de software a fazem informalmente e superficialmente, quando fazem. O tempo que se gasta identificando, analisando e controlando os riscos oferece retorno de muitas maneiras — menos pressão durante o projeto, uma melhor habilidade para acompanhar e controlar um projeto e a confiança que resulta do planejamento para os problemas antes que eles ocorram.

A análise de riscos pode absorver um volume significativo de trabalho de planejamento do projeto. A identificação, projeção, avaliação, gerenciamento e monitoração, tudo toma tempo. Mas o esforço é recompensado. Citando Sun Tzu, um general chinês que viveu há 2.500 anos, “Se você conhece o inimigo e conhece a si mesmo, não precisa temer o resultado de uma centena de batalhas”. Para o gerente de projeto de software, o inimigo é o risco.

PROBLEMAS E PONTOS A PONDERAR

- 28.1.** Forneça cinco exemplos de outros campos que ilustram os problemas associados à estratégia reativa de riscos.
- 28.2.** Descreva a diferença entre “riscos conhecidos” e “riscos previsíveis”.
- 28.3.** Acrescente três questões ou tópicos a cada uma das checklists de itens de risco apresentadas no site da SEPA.
- 28.4.** Você foi designado para criar um software que suporte um sistema de edição de vídeo de baixo custo. O sistema aceita como entrada vídeo digital, armazena vídeo em disco e depois permite que o usuário faça uma grande variedade de edições no vídeo digitalizado. O resultado pode então ser gravado em DVD ou outro tipo de mídia. Realize algumas pesquisas sobre sistemas desse tipo e depois faça uma lista dos riscos tecnológicos que enfrentaria ao empreender um projeto desse tipo.
- 28.5.** Você é o gerente de projeto de uma grande empresa de software. Foi designado para liderar uma equipe que está desenvolvendo um software processador de texto “avançado”. Crie uma tabela de riscos para o projeto.
- 28.6.** Descreva a diferença entre componentes de risco e fatores de risco.
- 28.7.** Desenvolva uma estratégia de mitigação de risco e especifique atividades específicas de mitigação de risco para três dos riscos descritos na Figura 28.2.
- 28.8.** Desenvolva uma estratégia de monitoração de risco e especifique as atividades de monitoramento de risco para três dos riscos descritos na Figura 28.2. Não se esqueça de identificar os fatores que estará monitorando para determinar se o risco está se tornando mais ou menos possível.
- 28.9.** Desenvolva uma estratégia de gerenciamento de risco e especifique atividades de gerenciamento de risco para três dos riscos descritos na Figura 28.2.
- 28.10.** Tente refinar três dos riscos descritos na Figura 28.2 e depois crie formulários de informações de risco para cada um deles.
- 28.11.** Represente três dos riscos mostrados na Figura 28.2 usando um formato CTC.
- 28.12.** Recalcule a exposição de risco discutida na Seção 28.4.2 quando o custo por LOC é de \$ 16 e a probabilidade é de 60%.
- 28.13.** Você pode pensar em uma situação na qual um risco de alta probabilidade e alto impacto não seria considerado como parte do seu plano RMMM?
- 28.14.** Descreva cinco áreas de aplicação de software nas quais a análise de segurança e imprevistos do software seriam uma preocupação importante.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

A literatura sobre gestão de risco de software se expandiu significativamente nas últimas décadas. Vun (*Modeling Risk*, Wiley, 2006) dá um tratamento matemático detalhado da análise de risco que pode ser aplicada aos projetos de software. Crohy e seus colegas (*The Essentials of Risk Management*, McGraw-Hill, 2006), Mulcahy (*Risk Management, Tricks of the Trade for Project Managers*, RMC Publications, Inc., 2003), Kendrick (*Identifying and Managing Project Risk*, American Management Association, 2003), e Marrison (*The Fundamentals of Risk Measurement*, McGraw-Hill, 2002) apresentam métodos úteis e ferramentas que todo gerente de projeto pode usar.

DeMarco e Lister (*Dancing with Bears*, Dorset House, 2003) escreveram um livro interessante e esclarecedor que orienta os gerentes e profissionais de software no gerenciamento de riscos. Moynihan (*Coping with IT/IS Risk Management*, Springer-Verlag, 2002) apresenta opiniões pragmáticas de gerentes de projeto que tratam do risco de forma contínua. Royer (*Project Risk Management*, Management Concepts, 2002) e Smith e Merritt (*Proactive Risk Management*, Productivity Press, 2002) sugerem um processo proativo para gestão de risco. Karolak (*Software Engineering Risk Management*, Wiley, 2002) escreveu um guia que introduz um modelo de análise de risco fácil de usar com checklists e questionários suportados por um pacote de software.

Capers Jones (*Assessment and Control of Software Risks*, Prentice Hall, 1994) apresenta uma discussão detalhada dos riscos de software, incluindo dados coletados de centenas de projetos de software. Jones define 60 fatores de risco que podem afetar o resultado dos projetos de software. Boehm [Boe89] sugere um excelente questionário e formatos de checklist que podem ser valiosos na identificação do risco. Charette [Cha89] apresenta uma abordagem detalhada da mecânica da análise de risco, usando teoria de probabilidades e técnicas estatísticas para analisar os riscos. Em outro volume, Charette (*Application Strategies for Risk Analysis*, McGraw-Hill, 1990) discute o risco no contexto de sistema e engenharia de software e sugere estratégias pragmáticas para gestão de risco. Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988) apresenta uma série de “princípios” (frequentemente são divertidos e às vezes profundos) que podem servir de excelente guia para gestão de risco.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects*, MIT Press, 2003) e Yourdon (*Death March*, Prentice Hall, 1997) discutem o que acontece quando os riscos engolfam a equipe de projeto de software. Bernstein (*Against the Gods*, Wiley, 1998) apresenta uma história interessante do risco que remonta aos tempos antigos.

O Software Engineering Institute publicou muitos relatórios e guias detalhados sobre análise e gestão de risco. O panfleto do Air Force Systems Command AFSCP 800-45 [AFC88] descreve técnicas de identificação e redução de riscos. Todas as edições do *ACM Software Engineering Notes* têm uma seção denominada “Risks to the Public” (editor, P. G. Neumann). Se você quiser conhecer as mais recentes e melhores histórias de horror do software, essa é a página a ser lida.

Há disponível na Internet uma grande variedade de fontes de informação sobre gestão de risco de software. Uma lista atualizada das referências da Web pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

análise de	
inventário.....	669
engenharia	
direta.....	675
engenharia	
reversa	670
dados.....	674
interface de	
usuário.....	673
processamento ..	672
manutenção de	
software	663
manutenibilidade..	664

Independentemente do domínio de aplicação, tamanho ou complexidade, o software continuará a evoluir com o tempo. As mudanças dirigem esse processo. No âmbito do software, ocorrem alterações quando são corrigidos erros, quando há adaptação a um novo ambiente, quando o cliente solicita novas características ou funções e quando a aplicação passa por um processo de reengenharia para proporcionar benefício em um contexto moderno. Durante os últimos 30 anos, Manny Lehman [por exemplo, Leh97a] e seus colegas executaram análises detalhadas do software industrial e sistemas para desenvolver a *teoria unificada para evolução do software* (*unified theory for software evolution*). Os detalhes desse trabalho estão além do escopo deste livro, mas vale a pena conhecer as leis que daí se originaram [Leh97b]:

A Lei da Mudança Contínua (1974): Softwares que foram implementados em um contexto de computação do mundo real e, portanto, irão evoluir com o tempo (chamados de *sistemas tipo E*) devem ser adaptados continuamente ou se tornarão cada vez menos satisfatórios.

A Lei da Complexidade Crescente (1974): À medida que um sistema tipo E evolui, sua complexidade aumenta, a menos que seja feito um trabalho para mantê-la ou reduzi-la.

PANORAMA

O que é? Considere qualquer produto de tecnologia que tenha lhe servido bem. Você o utiliza regularmente, mas ele está ficando obsoleto. Apresenta problemas com muita frequência, leva mais tempo para ser reparado do que você aceitaria e já não representa a tecnologia mais atualizada. O que fazer? Por um tempo, você tenta consertá-lo, remendá-lo ou até ampliar sua funcionalidade. Isso se chama manutenção. Mas a manutenção se torna cada vez mais difícil à medida que os anos passam. Chega então um momento em que precisa reformá-lo. Você criará um produto com mais funcionalidades, melhor desempenho e confiabilidade e de manutenção mais fácil. Isso é o que chamamos de reengenharia.

Quem realiza? No nível organizacional, a manutenção é executada por pessoal de suporte que faz parte da organização de engenharia de software. A reengenharia é executada por especialistas de negócio (muitas vezes empresas de consultoria), e no nível de software, a reengenharia é executada por engenheiros de software.

Por que é importante? Vivemos em um mundo que muda rapidamente. As demandas por funções de negócio e a tecnologia de informação que as suporta estão mudando em um ritmo que impõe enorme pressão competitiva em todas as organizações comerciais. É por essa razão que o software deve ser mantido continuamente. No momento apropriado, deve passar pela reengenharia para manter o ritmo.

Quais são as etapas envolvidas? A manutenção corrige defeitos, adapta o software para atender a um ambiente em mutação e melhora a funcionalidade para atender às necessidades dos clientes. Em um nível estratégico, a reengenharia de processos de negócio (*business process reengineering* – BPR) define objetivos comerciais, identifica e avalia processos comerciais existentes e cria processos de negócio revisados que melhor atendem aos objetivos atuais. A reengenharia de software abrange análise de inventário, reestruturação de documentos, engenharia reversa, reestruturação de programas e dados e engenharia direta. O objetivo dessas atividades é criar versões dos programas que apresentam uma qualidade mais alta e melhorar a manutenibilidade.

Qual é o artefato? É produzida uma variedade de produtos de manutenção e reengenharia (por exemplo, casos de uso, modelos de análise e projeto, procedimentos de teste). O resultado final é um software atualizado.

Como garantir que o trabalho foi realizado corretamente? Use as mesmas práticas de SQA aplicadas em todos os processos de engenharia de software — revisões técnicas avaliam os modelos de análise e projeto; revisões especializadas consideram a aplicabilidade e compatibilidade de negócio; e são aplicados testes para descobrir erros em conteúdo, funcionalidade e interoperabilidade.

reengenharia de processos de negócio (BPR).....	665
reestruturação dos documentos	669
reestruturação ...	674
código	674
dados.....	674
reengenharia de software	667
suportabilidade ...	664

 Como os sistemas legados evoluem com o tempo?

A Lei da Autorregulação (1974): O processo de evolução do sistema tipo E é autorregulado com distribuição do produto e medidas de processo próximo do normal.

A Lei da Conservação da Estabilidade Organizacional (1980): A taxa de atividade efetiva média em um sistema tipo E em evolução é invariante durante o tempo de vida do produto.

A Lei da Conservação da Familiaridade (1980): Conforme um sistema tipo E evolui, tudo o que está associado a ele, por exemplo, desenvolvedores, pessoal de vendas, usuários, deve manter o domínio de seu conteúdo e comportamento para uma evolução satisfatória. Um crescimento excessivo diminui esse domínio. Portanto, o crescimento incremental médio permanece invariante à medida que o sistema evolui.

A Lei do Crescimento Contínuo (1980): O conteúdo funcional dos sistemas tipo E deve ser continuamente ampliado durante toda a sua existência, para manter a satisfação do usuário.

A Lei da Qualidade em Declínio (1996): A qualidade dos sistemas tipo E irá parecer que está diminuindo a menos que sejam rigorosamente mantidos e adaptados às mudanças do ambiente operacional.

A Lei do Sistema de Realimentação (1996): Processos tipo E em evolução constituem sistemas de realimentação multinível, multilaço, multiagente e devem ser tratados como tais para que se obtenha uma melhora significativa em qualquer base razoável.

As leis que Lehman e seus colegas definiram são parte da realidade do engenheiro de software. Neste capítulo, discutiremos o desafio das atividades de manutenção e reengenharia de software necessárias para ampliar a vida efetiva dos sistemas legados.

29.1 MANUTENÇÃO DE SOFTWARE

A manutenção começa quase imediatamente. O software é liberado para os usuários finais, e em alguns dias, os relatos de bugs começam a chegar à organização de engenharia de software. Em algumas semanas, uma classe de usuários indica que o software deve ser mudado para se adaptar às necessidades especiais de seus ambientes. E em alguns meses, outro grupo corporativo, ainda não interessado no software quando foi lançado, agora reconhece que pode lhes trazer alguns benefícios. Eles precisarão de algumas melhorias para fazer o software funcionar em seu mundo.

O desafio da manutenção do software começou. Enfrentamos uma fila crescente de correções de bugs, solicitações de adaptação e melhorias que devem ser planejadas, programadas e, por fim, executadas. Logo, a fila já cresceu muito e o trabalho ameaça devorar os recursos disponíveis. Com o passar do tempo, sua organização descobre que está gastando mais tempo e dinheiro com a manutenção dos programas do que criando novas aplicações. De fato, não é raro uma organização de software despender de 60% a 70% de todos os recursos com manutenção de software.

Você talvez pergunte por que é necessário tanta manutenção e por que tanto esforço deve ser despendido. Osborne e Chikofsky [Osb90] fornecem uma resposta parcial:

Muitos softwares dos quais dependemos hoje têm em média de 10 a 15 anos. Mesmo quando esses programas foram criados, usando as melhores técnicas de projeto e codificação conhecidas na época [e muitos não foram], o tamanho do programa e o espaço de armazenamento eram as preocupações principais. Eles então migraram para novas plataformas, foram ajustados para mudanças nas máquinas e na tecnologia dos sistemas operacionais e aperfeiçoados para atender a novas necessidades dos usuários – tudo isso sem grande atenção na arquitetura geral. O resultado são estruturas mal projetadas, mal codificadas, de lógica pobre e mal documentadas em relação aos sistemas de software, para os quais somos chamados a fim de mantê-los rodando...

Outra razão para o problema de manutenção do software é a mobilidade dos profissionais. É bem provável que a equipe (ou profissional) responsável pelo trabalho original não esteja mais por perto. Pior ainda, outras gerações de profissionais de software podem ter modificado o sistema e já se foram. E hoje, pode não ter restado ninguém que tenha conhecimento direto do sistema legado.

“Manutenibilidade e clareza de programa são conceitos paralelos: quanto mais difícil for entender um programa, mais difícil será mantê-lo.

Gerald Berns

Conforme notamos no Capítulo 22, a natureza ubíqua das alterações permeia todo o trabalho de software. Mudanças são inevitáveis quando sistemas baseados em computadores são criados; portanto, você deve desenvolver mecanismos para avaliar, controlar e fazer modificações.

Em todo este livro, destacamos a importância de entender o problema (análise) e desenvolver uma solução bem estruturada (projeto). De fato, a Parte 2 deste livro é dedicada aos mecanismos dessas ações de engenharia de software e a Parte 3 concentra-se nas técnicas para garantir que você as tenha feito corretamente. Tanto a análise quanto o projeto levam a uma importante característica do software que chamamos de manutenibilidade, que, essencialmente, é uma indicação qualitativa¹ da facilidade com que o software pode ser corrigido, adaptado ou melhorado. Grande parte das funções da engenharia de software é criar sistemas que apresentem alta manutenibilidade.

Mas o que é manutenibilidade? Software “manutenível” apresenta uma modularidade eficaz (Capítulo 8). Utiliza padrões de projeto (Capítulo 12) que permitem entendê-lo facilmente. Foi construído usando padrões e convenções de codificação bem definidos, levando a um código-fonte autodocumentado e inteligível. Passou por uma variedade de técnicas de garantia de qualidade (Parte 3 deste livro) que descobriu potenciais problemas de manutenção antes que o software fosse lançado. Foi criado por engenheiros de software que reconhecem que não estarão por perto quando as alterações tiverem de ser feitas. Portanto, o projeto e a implementação do software deve “ajudar” a pessoa que for fazer a alteração.

29.2 SUPORTABILIDADE DO SOFTWARE

Para suportar efetivamente software de classe industrial, a organização (ou seus projetistas) deve ser capaz de fazer correções, adaptações e melhorias inerentes à atividade de manutenção. Além disso, a organização deve executar outras atividades importantes que incluem suporte operacional continuado, suporte ao usuário final e atividades de reengenharia durante toda a vida útil do software. Uma definição razoável da *suportabilidade do software* é

... a capacidade de suportar um sistema de software durante toda a vida útil do produto. Isso implica satisfazer quaisquer necessidades ou requisitos, mas também a provisão do equipamento, infraestrutura de suporte, software adicional, serviços de conveniências, mão de obra ou qualquer outro recurso necessário para manter o software operacional e capaz de satisfazer suas funções [SSO08].

Essencialmente, a suportabilidade é um dos muitos fatores de qualidade que devem ser considerados durante a análise e projeto na gestão da qualidade. Ela deve ser tratada como parte do modelo de requisitos (ou especificações) e considerada conforme o projeto evolui e a construção inicia.

Por exemplo, a necessidade de software “antidefeito” em nível de componente e código foi discutida anteriormente neste livro. O software deve conter facilidades para ajudar o pessoal de suporte quando encontrado um defeito no ambiente operacional (e não se iluda, eles *serão* encontrados). Além disso, o pessoal de suporte deve ter acesso a uma base de dados que contém os registros de todos os defeitos já detectados – suas características, causa e solução. Isso permitirá que o pessoal de suporte examine defeitos “similares” e possibilitará diagnóstico e correção mais rápidos.

Embora os erros encontrados em uma aplicação sejam um problema crítico de suporte, a suportabilidade também exige que sejam providenciados recursos para resolver os problemas diários dos usuários finais. A função do pessoal de suporte é responder às dúvidas dos usuários sobre instalação, operação e uso da aplicação.

WebRef

Uma grande quantidade de documentos sobre suportabilidade de software pode ser encontrada no site www.software-supportability.org/Downloads.html.

¹ Há algumas medidas quantitativas que fornecem uma indicação indireta da manutenibilidade (por exemplo, [Sch99], [SEI02]).

29.3 REENGENHARIA

Em um artigo seminal escrito para a *Harvard Business Review*, Michael Hammer [Ham90] lançou os fundamentos para uma revolução no modo de pensar dos gerentes sobre processos de negócio e computação:

Está na hora de parar de seguir as trilhas das vacas. Em vez de acrescentar processos ultrapassados em hardware e software, devemos rejeitá-los e começar de novo. Devemos fazer uma “reengenharia” em nossos negócios: usar o poder da moderna tecnologia de informação para redesenhar radicalmente nossos processos de negócio para obter melhoras significativas no desempenho.

Toda empresa opera de acordo com muitas regras não articuladas... A reengenharia luta para romper com as velhas regras sobre como organizamos e conduzimos nossos negócios.

“Encarar o amanhã pensando em usar métodos de ontem é ver a vida estagnada.”

James Bell

Como ocorre em todas as revoluções, a agitação provocada por Hammer resultou em mudanças tanto positivas quanto negativas. Durante a década de 1990, algumas empresas fizeram um legítimo esforço para a reengenharia, e os resultados levaram a uma melhor competitividade. Outras ficaram apenas no downsizing e na terceirização (em vez da reengenharia) para melhorar sua sobrevivência. Isso resultou no aparecimento de organizações “medianas” com pouco potencial para crescimento futuro [DeM95a].

No fim da primeira década do século 21, a moda associada à reengenharia desapareceu, mas o processo continua por si só em grandes e pequenas empresas. A ligação entre reengenharia nos negócios e reengenharia no software baseia-se em uma “visão de sistema”.

O software é muitas vezes a realização das regras de negócio discutidas por Hammer. Hoje, grandes empresas possuem dezenas de milhares de programas de computador que suportam as “velhas regras de negócio”. Enquanto os gerentes trabalham para modificar as regras para obter maior eficiência e competitividade, o software deve acompanhar. Em alguns casos, isso significa a criação de grandes sistemas baseados em computadores.² Mas em alguns outros, significa a modificação ou reforma de aplicações existentes.

Nas seções a seguir, examinaremos a reengenharia de um modo descendente, começando com uma rápida visão geral da reengenharia de processos de negócio passando para uma discussão mais detalhada das atividades técnicas que ocorrem quando o software sofre uma reengenharia.

29.4 REENGENHARIA DE PROCESSO DE NEGÓCIO

PONTO-CHAVE

A reengenharia de processos de negócio (BPR) muitas vezes resulta em nova funcionalidade de software, enquanto a reengenharia de software trabalha para substituir funcionalidade existente por um software melhor, mais fácil de manter.

A reengenharia de processos de negócio (*business process reengineering* — BPR) se estende muito além do escopo das tecnologias de informação e da engenharia de software. Entre as muitas definições (muitas um tanto abstratas) sugeridas para a BPR, destaca-se a publicada na revista *Fortune Magazine* [Ste93]: “a busca para, e a implementação de, mudanças radicais nos processos de negócio para obter resultados excelentes”. Mas como é feita a busca e como é obtida a implementação? Mais importante ainda, como podemos assegurar que a “mudança radical” sugerida levará de fato a “resultados inovadores” e não ao caos organizacional?

29.4.1 Processos de negócio

Um processo de negócio é “um conjunto de tarefas logicamente relacionadas executadas para obter um resultado de negócio definido” [Dav90]. No processo de negócio, pessoas, equipamentos, recursos materiais e procedimentos são combinados para produzir um resultado específico. Exemplos de processos de negócio incluem projeto de um novo produto, compra de serviços e suprimentos, contratação de funcionários e pagamento dos fornecedores. Cada um desses demanda uma série de tarefas e cada um utiliza diversos recursos na empresa.

2 A explosão de aplicações e sistemas baseados na Web é uma indicação dessa tendência.



Como engenheiro de software, seu trabalho ocorre na base dessa hierarquia. Porém, você deve se certificar de que alguém tenha dado séria atenção aos níveis acima. Se isso não foi feito, seu trabalho está em risco.

Todo processo de negócio tem um cliente definido — uma pessoa ou grupo que recebe o resultado (por exemplo, uma ideia, um relatório, um projeto, um serviço, um produto). Além disso, os processos de negócio cruzam as fronteiras da organização. Requerem que diferentes grupos organizacionais participem das “tarefas logicamente relacionadas” que definem o processo.

Cada sistema é na realidade uma hierarquia de subsistemas. Um negócio não é uma exceção; de forma geral é segmentado da seguinte maneira:

O negócio → sistemas de negócio → processos de negócio → subprocessos de negócio

Cada sistema de negócio (também chamado de *funções de negócio*) é composto de um ou mais processos, e cada processo de negócio é definido por uma série de subprocessos.

A BPR pode ser aplicada a qualquer nível da hierarquia, à medida que o escopo da BPR se amplia (conforme subimos na hierarquia), os riscos associados a ela crescem significativamente. Por essa razão, muitos esforços de BPR concentram-se em processos individuais e subprocessos.

29.4.2 Um modelo de BPR

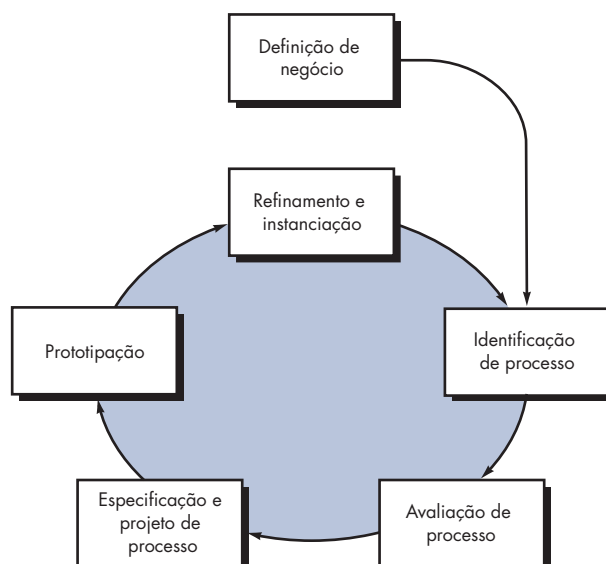
Assim como a maioria das atividades de engenharia, a reengenharia dos processos de negócio é iterativa. As metas de negócio e os processos para atingi-las devem ser adaptados a um ambiente de negócio em mutação. Por essa razão, não há um início e fim para a BPR — é um processo evolucionário. Um modelo para reengenharia de processo de negócio é apresentado na Figura 29.1. O modelo define seis atividades:

Definição de negócio. As metas dos negócios são identificadas em um contexto dos quatro motivadores principais: redução de custo, redução do tempo, melhoria da qualidade e desenvolvimento pessoal. As metas podem ser definidas no nível do negócio ou para um componente específico do negócio.

Identificação do processo. São identificados os processos críticos para atingir as metas estabelecidas na definição do negócio. Eles podem então ser classificados por importância, por necessidade de mudança ou de qualquer outra forma apropriada para a atividade de reengenharia.

Avaliação de processo. O processo existente é amplamente analisado e medido. Identificam-se as tarefas do processo, registram-se os custos e o tempo consumido pelas tarefas do processo, e são isolados os problemas de qualidade/desempenho.

FIGURA 29.1
Um modelo de BPR



“Tão logo identificamos algo conhecido em uma coisa nova, nos tranquilizamos.”

F. W. Nietzsche

Especificação e projeto do processo. Com base nas informações obtidas durante as três primeiras atividades da BPR, são preparados casos de uso (Capítulos 5 e 6) para cada processo que deve ser reprojetoado. No contexto da BPR, os casos de uso identificam um cenário que fornece algum resultado a um cliente. Com o caso de uso como especificação do processo, um novo conjunto de tarefas é projetado para o processo.

Protótipo. Um processo de negócio reprojetoado deve passar por um protótipo antes de ser totalmente integrado nos negócios. Essa atividade “testa” o processo de modo que possam ser feitos os refinamentos.

Refinamento e instanciação. Com base nas informações obtidas do protótipo, o processo de negócio é refinado e instanciado em um sistema de negócio.

As atividades de BPR são às vezes utilizadas em conjunto com ferramentas de análise de fluxo de trabalho. A finalidade das ferramentas é criar um modelo do fluxo de trabalho existente, em um esforço para melhor analisar os processos.

29.5 REENGENHARIA DE SOFTWARE

O cenário é muito comum. Um aplicativo atendeu às necessidades de negócio de uma empresa por 10 ou 15 anos. Durante esse tempo, ele foi corrigido, adaptado e aperfeiçoado muitas vezes. Profissionais realizaram esse trabalho com as melhores intenções, mas as boas práticas de engenharia de software foram sempre deixadas de lado (devido à pressão por outros aspectos). Agora o aplicativo está instável. Ainda funciona, mas sempre que se tenta fazer uma alteração, ocorrem efeitos colaterais sérios e inesperados. No entanto, o aplicativo deve continuar evoluindo. O que fazer?

Software que não pode ser mantido não é novidade. Na verdade, a ênfase cada vez mais maior sobre a reengenharia de software foi motivada pelos problemas de manutenção criados por mais de quatro décadas.

FERRAMENTAS DO SOFTWARE



Reengenharia de processos de negócio (business process reengineering-BPR)

Objetivo: o objetivo das ferramentas de BPR é suportar a análise e avaliação de processos de negócio existentes e a especificação de projetos de novos.

Mecânica: a mecânica das ferramentas varia. Em geral, as ferramentas de BPR permitem que um analista modele os processos de negócio existentes em um esforço para avaliar as ineficiências do fluxo de trabalho ou problemas funcionais. Uma vez identificados os problemas, existem ferramentas que permitem a análise do protótipo e/ou simulação de processos de negócio revisados.

Ferramentas representativas:³

Extend, desenvolvida pela ImagineThat, Inc. (www.imagine-that.com), é uma ferramenta de simulação para modelar processos existentes e explorar novos processos. A ferramenta Extend proporciona um recurso amplo do tipo “o

que-se” que permite a um analista de negócio explorar diferentes cenários de processo.

e-Work, desenvolvida pela Metastorm (www.metastorm.com), proporciona suporte de gerenciamento de processos de negócio tanto para processos manuais quanto automáticos.

IceTools, desenvolvida pela Blue Ice (www.blueice.com), é uma coleção de modelos de BPR para o Microsoft Office e Microsoft Project.

SpeedDev, desenvolvida pela SpeedDev Inc. (www.speeddev.com), é uma das muitas ferramentas que permitem a uma organização modelar o fluxo de trabalho de processo (nesse caso, fluxo de trabalho de TI).

Workflow tool suite, desenvolvida pela MetaSoftware (www.metasoftware.com), incorpora um conjunto de ferramentas para modelagem de fluxo de trabalho, simulação e cronogramas.

Uma boa lista de links de ferramentas de BPR pode ser encontrada no site www.opfro.org/index.html?Components/Producers/Tools/BusinessProcessReengineeringTools.html~Contents.

³ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

29.5.1 Um modelo de processo de reengenharia de software

Reengenharia toma tempo, tem um custo significativo em dinheiro e absorve recursos que poderiam de outra forma ser usados em necessidades mais imediatas. Por todas essas razões, a reengenharia não é realizada em alguns meses ou mesmo anos. A reengenharia dos sistemas de informação é uma atividade que absorverá recursos da tecnologia de informação por muitos anos. Todas as organizações precisam de uma estratégia pragmática para reengenharia de software.

Uma estratégia prática faz parte de um modelo de processo de reengenharia. Discutiremos o modelo mais tarde nesta seção, mas primeiro, vejamos alguns princípios básicos.

A reengenharia é um trabalho de reforma. Para melhor entendê-la, considere uma atividade análoga: a reforma de uma casa. Imagine a seguinte situação. Você acaba de comprar uma casa em outro Estado, mas nunca viu o imóvel. Comprou-o por um preço extremamente baixo, tendo sido alertado de que a casa poderia necessitar de reforma completa. Como você faria?

- Antes de iniciar a reforma, seria razoável inspecionar a casa. Para determinar se precisa de reforma, você (ou um profissional de construção) criaria uma lista de critérios para que a inspeção fosse sistemática.
- Antes de começar a reconstrução, verifique como está a estrutura. Se a casa estiver com a estrutura em bom estado, pode ser possível “remodelar” sem reformar (a um custo bem mais baixo e em menos tempo).
- Antes de começar a reforma, procure entender como a casa original foi construída. Dê uma olhada atrás das paredes. Verifique a fiação elétrica, a tubulação hidráulica e as partes internas da estrutura. Mesmo que você resolva descartar tudo, as informações obtidas terão utilidade quando iniciar a reconstrução.
- Na reforma, use somente os materiais mais modernos e mais duráveis. Isso pode custar um pouco mais agora, mas ajudará a evitar uma manutenção cara e demorada mais tarde.
- Se você decide reformar, seja disciplinado. Use práticas que resultarão na mais alta qualidade — hoje e no futuro.

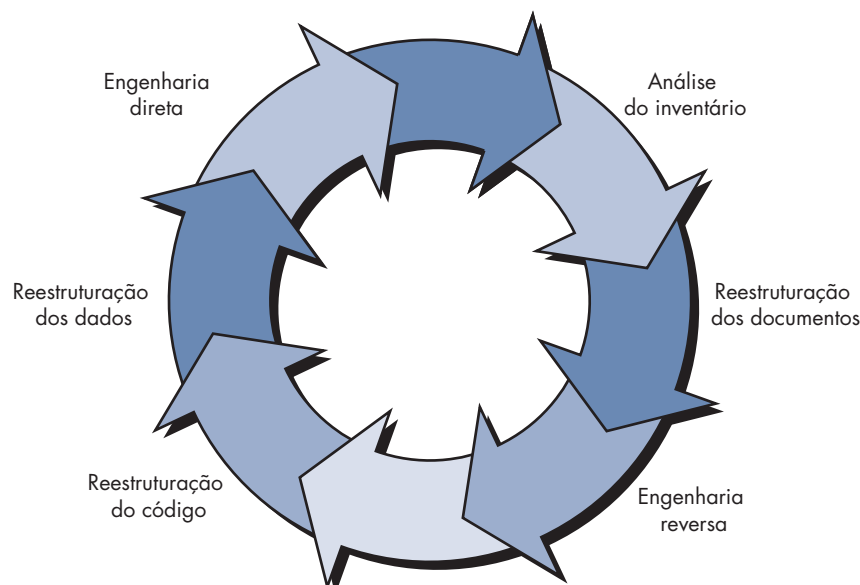
Embora esses princípios concentrem-se na reforma de uma casa, eles se aplicam igualmente bem à reengenharia de sistemas e aplicações baseados em computador.

Para implementarmos esses princípios, podemos usar um modelo de processo de reengenharia de software, apresentado na Figura 29.2. Em alguns casos, essas atividades ocorrem em

WebRef

Uma excelente fonte de informações sobre reengenharia de software pode ser encontrada no site reengineer.org.

FIGURA 29.2
Um modelo de processo de reengenharia de software



sequência linear, mas nem sempre é o caso. Por exemplo, pode acontecer de a engenharia reversa (entendimento do funcionamento interno de um programa) ter de ocorrer antes do início da reestruturação dos documentos.



Se o tempo e os recursos estiverem escassos, você pode aplicar o princípio de Pareto ao software que vai passar por reengenharia. Aplique o processo de reengenharia aos 20% do software, responsáveis por 80% dos problemas.



Crie apenas a documentação necessária para melhor entender o software, nem uma página a mais.

29.5.2 Atividades de reengenharia de software

O paradigma da reengenharia mostrado na Figura 29.2 é um modelo cíclico. Isso significa que cada uma das atividades apresentadas como parte do paradigma pode ser revisitada. Para qualquer ciclo em particular, o processo pode terminar após qualquer uma dessas atividades.

Análise de inventário. Toda organização de software deve ter um inventário de todos os aplicativos. O inventário pode ser nada mais do que uma planilha com informações detalhadas (por exemplo, tamanho, idade, criticalidade nos negócios) para cada aplicativo ativo. Ordenando essas informações de acordo com a criticalidade de negócio, longevidade, manutenibilidade atual e suportabilidade, e outros critérios localmente importantes, surgem os candidatos à reengenharia. Recursos podem então ser alocados aos aplicativos candidatos ao trabalho de reengenharia.

É importante observar que o inventário deverá ser revisto regularmente. O status dos aplicativos (por exemplo, criticalidade de negócio) pode mudar em função do tempo, e, como resultado, as prioridades para a reengenharia também podem mudar.

Reestruturação dos documentos. Documentação pobre é a marca registrada de muitos sistemas legados. O que se pode fazer quanto a isso? Quais são as suas opções?

1. *Criar documentação consome muito tempo.* Se o sistema funciona, você pode optar por conviver com aquilo que tem. Em alguns casos, essa é a atitude correta. Não é possível recriar documentação para centenas de programas de computador. Se um programa for relativamente estático, está chegando ao fim de sua vida útil, e provavelmente não terá uma modificação significativa, deixe-o como está!
2. *A documentação tem de ser atualizada, mas sua organização apresenta recursos limitados.* Você empregará uma abordagem “documentar quando usar”. Pode não ser necessário redocumentar totalmente o aplicativo. Em vez disso, aquelas partes do sistema que estão passando por alteração são completamente documentadas. Com o tempo, você terá uma coleção de documentos úteis e importantes.
3. *O sistema é crítico para os negócios e deve ser totalmente documentado.* Mesmo neste caso, uma abordagem inteligente é limitar a documentação a um mínimo essencial.

Cada uma dessas opções é viável. Sua organização de software deve escolher a mais apropriada para cada caso.

Engenharia reversa. O termo *engenharia reversa* tem suas origens no mundo do hardware. Uma empresa desmonta um produto de hardware competitivo na tentativa de conhecer os “segredos” de projeto e fabricação do concorrente. Os segredos poderiam ser facilmente entendidos se fosse possível obter as especificações de projeto e fabricação do concorrente. Mas esses documentos são de propriedade privada e não estão disponíveis para a empresa que está fazendo a engenharia reversa. Essencialmente, uma engenharia reversa bem-sucedida resulta em uma ou mais especificações de projeto e fabricação para um produto pelo exame de amostras atuais do produto.

A engenharia reversa para o software é bem similar. Em muitos casos, no entanto, o programa a ser submetido a uma engenharia reversa não é o programa de um concorrente. Em vez disso, é o próprio trabalho da empresa (em geral, feito há muitos anos). Os “segredos” a ser entendidos são obscuros porque nenhuma especificação jamais foi desenvolvida. Portanto, a engenharia reversa para software é o processo para analisar um programa na tentativa de criar uma representação do programa em um nível mais alto de abstração do que o código-fonte.

WebRef

Um conjunto de recursos para a comunidade de reengenharia pode ser obtido no site www.comp.lancs.ac.uk/projects/RenaissanceWeb/.

A engenharia reversa é um processo de *recuperação do projeto*. As ferramentas de engenharia reversa extraem informações do projeto de dados, da arquitetura e procedural com base em um programa existente.

Reestruturação do código. O tipo mais comum de reengenharia (atualmente, o uso do termo reengenharia é questionável neste caso) é a *reestruturação de código*.⁴ Alguns sistemas legados têm uma arquitetura de programa razoavelmente sólida, mas os módulos individuais foram co-dificados de um modo que se torna difícil de entendê-los, testá-los e mantê-los. Nesses casos, o código dentro dos módulos suspeitos pode ser reestruturado.

Para executar essa atividade, o código-fonte é analisado por meio de uma ferramenta de reestruturação. As violações das construções de programação estruturada são registradas, e o código é então reestruturado (isso pode ser feito automaticamente) ou mesmo reescrito em uma linguagem de programação mais moderna. O código reestruturado resultante é revisado e testado para garantir que não tenham sido introduzidas anomalias. A documentação interna do código é atualizada.

Reestruturação dos dados. Um programa com uma arquitetura de dados fraca será difícil de adaptar e melhorar. Na verdade, para muitas aplicações, a arquitetura das informações tem mais a ver com a viabilidade de longo prazo de um programa do que o próprio código-fonte.

Diferentemente da reestruturação de código, que ocorre em um nível relativamente baixo de abstração, a reestruturação de dados é uma atividade de reengenharia em escala completa. Em muitos casos, a reestruturação dos dados começa com uma atividade de engenharia reversa. A arquitetura de dados atual é dissecada, e os modelos de dados necessários são definidos (Capítulos 6 e 9). Identificam-se os objetos de dados e atributos, e as estruturas de dados existentes são revisadas quanto à qualidade.

Quando a estrutura de dados é fraca (por exemplo, estão implementados em arquivos de texto, quando uma abordagem relacional simplificaria muito o processamento), os dados passam por uma reengenharia.

Devido à arquitetura de dados ter uma forte influência sobre a arquitetura do programa e os algoritmos que a constituem, mudanças nos dados resultarão invariavelmente em mudanças de arquitetura ou de nível de código.

Engenharia direta. Em um mundo ideal, os aplicativos seriam recriados por meio de um “motor de reengenharia” automatizado. O programa antigo seria colocado nesse motor, analisado, reestruturado e regenerado em uma forma que apresentasse os melhores aspectos da qualidade do software. Em poucas palavras, é improvável que tal “motor” algum dia apareça, mas os fabricantes de software introduziram ferramentas que fornecem um subconjunto limitado desses recursos que se destinam a domínios de aplicação específicos (por exemplo, aplicações implementadas por meio de um sistema de base de dados específico). Mais importante, essas ferramentas de reengenharia estão se tornando cada vez mais sofisticadas.

A engenharia direta não apenas recupera as informações do projeto do software existente, mas também usa as informações para alterar ou reconstituir o sistema existente em um esforço para melhorar sua qualidade geral. Em muitos casos, o software que passou pela reengenharia reimplementa a função do sistema existente e também acrescenta novas funções e/ou melhora o desempenho geral.

29.6 ENGENHARIA REVERSA

A engenharia reversa evoca uma imagem da “fenda mágica”. Você coloca na fenda uma listagem de código não documentada, criada de qualquer jeito, e do outro lado sai uma descrição completa de

⁴ A reestruturação de código tem alguns dos elementos de “refabricação”, um conceito de redesenho introduzido no Capítulo 8 e discutido em outras partes deste livro.

projeto (e documentação completa) para o programa do computador. Infelizmente, a fenda mágica não existe. A engenharia reversa pode extrair informações do projeto do código-fonte, mas o nível de abstração, a completeza da documentação, o grau segundo o qual as ferramentas e um analista humano trabalham juntos e a direcionalidade do processo são altamente variáveis.

O *nível de abstração* de um processo de engenharia reversa e as ferramentas utilizadas para realizá-lo referem-se à sofisticação das informações de projeto que podem ser extraídas do código-fonte. Idealmente, o nível de abstração deverá ser tão alto quanto possível. Isto é, o processo de engenharia reversa deverá ser capaz de derivar representações de projeto procedural (em uma abstração de baixo nível), informações de programa e de estrutura de dados (em um nível de abstração um tanto mais alto), modelos de objeto, dados e/ou modelos de fluxo de controle (em um nível de abstração relativamente alto) e modelos de entidade-relacionamento (em um nível alto de abstração). À medida que o nível de abstração aumenta, você recebe informações que lhe permitirão entender o programa mais facilmente.

A *completeza* de um processo de engenharia reversa refere-se ao nível de detalhe fornecido em um nível de abstração. Em muitos casos, a completeza diminui conforme o nível de abstração aumenta. Por exemplo, dada uma listagem de código-fonte, é relativamente fácil desenvolver uma representação completa do projeto procedural. Podem ser derivadas também representações simples de projeto arquitetônico, mas é muito mais difícil desenvolver um conjunto completo de diagramas UML ou modelos.

A completeza melhora em proporção direta com a quantidade de análise executada pela pessoa que está fazendo a engenharia reversa. *Interatividade* refere-se ao grau segundo o qual as pessoas são “integradas” com as ferramentas automáticas para criar um processo eficaz de engenharia reversa. Em muitos casos, na medida em que o nível de abstração aumenta, a interatividade deve aumentar ou a completeza será prejudicada.

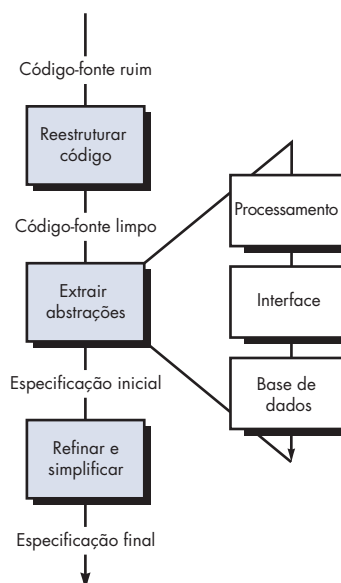
Se a *direcionalidade* do processo de engenharia reversa for um único sentido, todas as informações extraídas do código-fonte serão fornecidas ao engenheiro de software que pode então usá-las durante qualquer atividade de manutenção. Se a direcionalidade for nos dois sentidos, as informações serão colocadas em uma ferramenta de reengenharia que tenta reestruturar ou regenerar o sistema antigo.

O processo de engenharia reversa está representado na Figura 29.3. Antes de começar as atividades de engenharia reversa, o código-fonte não estruturado (“ruim”) é reestruturado (Seção 29.5.1)

WebRef

Recursos úteis para “recuperação de design e entendimento de programa” podem ser encontrados no site www.sel.iit.nrc.ca/projects/dr/dr.html.

FIGURA 29.3
O processo de engenharia reversa



para que contenha somente as construções de programação estruturada.⁵ Isso torna o código-fonte mais fácil de ler e proporciona a base para todas as atividades subsequentes de engenharia reversa.

O núcleo da engenharia reversa é uma atividade chamada de *extração de abstrações*. Você deve avaliar o programa antigo e, com base no código-fonte (muitas vezes não documentado), desenvolver uma especificação significativa do processamento executado, da interface de usuário aplicada e das estruturas de dados de programas ou base de dados utilizada.

29.6.1 Engenharia reversa para entender os dados

A engenharia reversa dos dados ocorre em diferentes níveis de abstração e frequentemente é a primeira tarefa da reengenharia. No nível de programa, as estruturas internas de dados de programa devem muitas vezes passar por uma engenharia reversa como parte de um trabalho de reengenharia total. Em nível de sistema, estruturas de dados globais (por exemplo, arquivos, bases de dados) passam muitas vezes por uma reengenharia para acomodar novos paradigmas de gerenciamento de base de dados (por exemplo, a mudança de arquivos de texto para sistemas de bases de dados relacionais ou orientados a objeto). A engenharia reversa das estruturas de dados globais atuais define o cenário para a introdução de uma nova base de dados para todo o sistema.



Em alguns casos, a primeira atividade de reengenharia tenta construir um diagrama de classe UML.



A abordagem para a engenharia reversa de dados para software convencional segue um caminho análogo: (1) criar um modelo de dados, (2) identificar atributos dos objetos dados, e (3) definir relações.

Estruturas internas de dados. As técnicas de engenharia reversa para dados internos de programa focalizam a definição de classes de objetos. Isso é feito examinando-se o código do programa para agrupar variáveis de programa relacionadas. Em muitos casos, a organização de dados dentro do código identifica tipos de dados abstratos. Por exemplo, estruturas de registro, arquivos, listas e outras estruturas de dados muitas vezes fornecem um indicador inicial das classes.

Estrutura de base de dados. Independentemente de sua organização lógica e estrutura física, uma base de dados permite a definição de objetos de dados e suporta algum método para estabelecer relações entre os objetos. Portanto, para fazer a reengenharia de um esquema de base de dados para outro é necessário entender os objetos e suas relações.

Podem ser usados os passos a seguir [Pre94] para definir o modelo de dados existente como um precursor para a reengenharia de um novo modelo de base de dados: (1) criar um modelo do objeto inicial, (2) determinar chaves candidatas (são examinados os atributos para determinar se elas são usadas para apontar para outro registro ou tabela; aquelas que servem como ponteiros tornam-se chaves candidatas), (3) refinar as classes provisórias, (4) definir generalizações, e (5) descobrir associações usando técnicas análogas à abordagem CRC. Uma vez conhecidas as informações definidas nos passos anteriores, pode ser aplicada uma série de transformações [Pre94] para mapear a estrutura antiga de base de dados em uma nova estrutura de base de dados.

29.6.2 Engenharia reversa para entender o processamento

A engenharia reversa para entender o processamento começa com uma tentativa de entender e extrair abstrações procedurais representadas pelo código-fonte. Para entender as abstrações procedurais, o código é analisado em vários níveis de abstrações: sistema, programa, componente, padrão e instruções.

A funcionalidade global de todo o sistema da aplicação deve ser entendida antes do trabalho detalhado de engenharia reversa. Isso estabelece um contexto para análise posterior e proporciona uma visão sobre os problemas de interoperabilidade entre os aplicativos no sistema. Cada um dos programas que formam o sistema de um aplicativo representa uma abstração funcional em um alto nível de detalhe. É criado um diagrama de blocos, representando a interação entre essas abstrações funcionais. Cada componente executa alguma subfunção e representa uma abstração procedural definida. É desenvolvida uma narrativa de processamento para cada componente. Em algumas situações, já existem especificações de sistema, programa e componente.

“Existe uma paixão pela compreensão, assim como uma paixão pela música. Essa paixão é muito comum nas crianças, mas em muitas pessoas acaba se perdendo mais tarde.”

Albert Einstein

5 O código pode ser reestruturado usando um *motor de reestruturação* — uma ferramenta que reestrutura código-fonte.

Quando esse é o caso, as especificações são revistas para verificar a conformidade com o código existente.⁶

A situação torna-se mais complexa quando é considerado o código dentro de um componente. Você deverá procurar seções de código que representam padrões procedurais genéricos. Em quase todos os componentes, uma seção do código prepara dados para processamento (dentro do módulo), outra seção faz o processamento e outra prepara os resultados do processamento para exportação do componente. Em cada uma dessas seções, podem-se encontrar padrões menores; por exemplo, validação de dados e verificações de limites, que muitas vezes ocorrem na seção do código que prepara os dados para processamento.

Para sistemas grandes, a engenharia reversa em geral é feita usando-se uma abordagem semiautomática. Podem ser empregadas ferramentas automatizadas para ajudá-lo a entender a semântica do código existente. O resultado desse processo é então passado para as ferramentas de reestruturação e engenharia direta para completar o processo de reengenharia.

29.6.3 Engenharia reversa das interfaces de usuário

As GUIs sofisticadas tornaram-se uma exigência para produtos e sistemas de todos os tipos baseados em computadores. Portanto, o desenvolvimento de interfaces de usuário tornou-se um dos tipos mais comuns de atividade de reengenharia. Mas antes de recriar uma interface de usuário, deverá ocorrer a engenharia reversa.

Para entender completamente uma interface de usuário, deve ser especificada a estrutura e o comportamento da interface. Merlo e seus colegas [Mer93] sugerem três questões básicas que devem ser respondidas quando se começa a engenharia reversa da interface de usuário (UI):

- Quais são as ações básicas (por exemplo, teclas e cliques de mouse) que a interface deve processar?
- Qual é a descrição compacta da resposta comportamental do sistema para essas ações?
- O que significa “substituição” ou, mais precisamente, que conceito de equivalência de interfaces é relevante aqui?

A notação de modelagem comportamental (Capítulo 7) pode proporcionar um meio para desenvolver respostas para as duas primeiras questões. Grande parte das informações necessárias para criar um modelo comportamental pode ser obtida observando-se a manifestação externa da interface. Mas as informações adicionais necessárias para criar o modelo comportamental devem ser extraídas do código.

? Como podemos entender o funcionamento de uma interface de usuário existente?

FERRAMENTAS DO SOFTWARE



Engenharia reversa

Objetivo: ajudar os engenheiros de software a entender a estrutura de projeto interna de programas complexos.

Mecânica: em muitos casos, as ferramentas de engenharia reversa aceitam código-fonte como entrada e produzem uma variedade de representações de projeto estrutural, procedural, dados e comportamental.

Ferramentas representativas:⁷

Imagix 4D, desenvolvida pela Imagix (www.imagix.com), “ajuda os desenvolvedores de software a entender software

em C e C++ complexo ou legado” pela engenharia reversa e documentação do código-fonte.

Understand, desenvolvida pela Scientific Toolworks, Inc. (www.scitools.com), analisa Ada, Fortran, C, C++ e Java “para fazer a engenharia reversa, documentar automaticamente, calcular métricas de código e ajudá-lo a entender, navegar e manter o código-fonte”.

Pode ser encontrada uma lista abrangente de ferramentas para engenharia reversa no site <http://scgwiki.iam.unibe.ch:8080/SCG/370>.

⁶ Frequentemente, especificações escritas no início da vida do programa nunca são atualizadas. À medida que são feitas as atualizações, o código não está mais em conformidade com a especificação.

⁷ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

É importante notar que uma GUI substituta pode não refletir exatamente a interface antiga (na verdade, pode ser radicalmente diferente). Muitas vezes compensa desenvolver uma nova metáfora de interação. Por exemplo, uma interface de usuário antiga requer que o usuário forneça um fator de escala (variando de 1 a 10) para reduzir ou ampliar uma imagem gráfica. Uma GUI que passou pela reengenharia pode usar uma barra de rolagem e o mouse para executar a mesma função.

29.7 REESTRUTURAÇÃO

A reestruturação de software modifica o código-fonte e/ou os dados para torná-lo mais amigável para futuras alterações. Em geral, a reestruturação não modifica a arquitetura geral do programa. Ela tende a concentrar-se nos detalhes de projeto dos módulos individuais e nas estruturas de dados locais definidas nos módulos. Se o trabalho de reestruturação se estende além dos limites de módulo e abrange a arquitetura do software, a reestruturação passa a ser engenharia direta (Seção 29.7).



Embora a reestruturação de código possa aliviar problemas imediatos associados com o debugging ou pequenas alterações, ela não é a reengenharia. O verdadeiro benefício é conseguido somente quando os dados e a arquitetura são reestruturados.

A reestruturação ocorre quando a arquitetura básica de um aplicativo é sólida, mesmo que as partes técnicas internas necessitem de um retrabalho. Ela ocorre quando partes importantes do software são reparáveis e somente um subconjunto de todos os módulos e dados necessita de uma modificação mais extensa.⁸

29.7.1 Reestruturação de código

A *reestruturação de código* é feita para obter um projeto que produz a mesma função, mas com mais qualidade do que o programa original. Em geral, as técnicas de reestruturação de código (por exemplo, as técnicas de simplificação lógica de Warnier [War74]) modelam a lógica de programação usando álgebra booleana e aplicam uma série de regras de transformação que resulta na lógica reestruturada. O objetivo é pegar um código “emaranhado” e obter um projeto procedural que esteja em conformidade com a filosofia de programação estruturada (Capítulo 10).

Outras técnicas de reestruturação também foram propostas para uso com as ferramentas de reengenharia. Um diagrama de intercâmbio de recursos mapeia cada módulo de programa e os recursos (tipos de dados, procedimentos e variáveis) permutados entre ele e outros módulos. Criando representações do fluxo de recursos, a arquitetura do programa pode ser reestruturada para obter o mínimo acoplamento entre os módulos.

29.7.2 Reestruturação de dados

Antes de iniciar a reestruturação de dados, deve ser feita uma atividade de engenharia reversa chamada de *análise do código-fonte*. São avaliadas todas as instruções da linguagem de programação que contenham definições de dados, descrições de arquivo, I/O e descrições de interface. A finalidade é extrair itens de dados e objetos, para obter informações sobre fluxo de dados e para entender as estruturas de dados existentes que precisam ser implementadas. Essa atividade às vezes é chamada de *análise de dados*.

Uma vez completada a análise de dados, inicia-se o *reprojeto dos dados*. Em sua forma mais simples, uma etapa de *padronização de registro de dados* esclarece as definições de dados para obter consistência entre nomes de itens de dados ou formatos de registros físicos em uma estrutura de dados ou formato de arquivo existentes. Outra forma de reprojeto, chamada de *racionalização de nomes de dados*, garante que todas as convenções de nomes de dados estejam de acordo com os padrões locais e que os pseudônimos (alias) sejam eliminados à medida que os dados fluem através do sistema.

Quando a reestruturação vai além da padronização e racionalização, são feitas modificações físicas nas estruturas de dados existentes para tornar mais eficaz o projeto de dados. Isso pode

⁸ Às vezes é difícil fazer a distinção entre reestruturação extensiva e redesevolvimento. Ambos os casos são reengenharia.

significar a transformação de um formato de arquivo para outro ou, em alguns casos, a transformação de um tipo de base de dados para outro.

FERRAMENTAS DO SOFTWARE



Reestruturação de software

Objetivo: o objetivo das ferramentas de reestruturação é transformar software não estruturado e mais antigo em linguagens de programação e estruturas de projeto modernas.

Mecânica: em geral, o código-fonte é o elemento de entrada e ele é transformado em um programa mais bem estruturado. Em alguns casos, a transformação ocorre na mesma linguagem de programação. Em outros, uma linguagem de programação mais antiga é transformada em uma mais moderna.

Ferramentas representativas:⁹

DMS Software Reengineering Toolkit, desenvolvida pela Semantic Design (www.semdesigns.com), fornece uma variedade de recursos de reestruturação para COBOL, /C++, Java, Fortran 90 e VHDL.

Clone Doctor, desenvolvida pela Semantic Designs, Inc. (www.semdesigns.com), analisa e transforma programas escritos em C, C++, Java ou COBOL ou qualquer outra linguagem de programação de computador baseada em texto.

plusFORT, desenvolvida pela Polyhedron (www.polyhedron.com), é um conjunto de ferramentas FORTRAN contendo recursos para reestruturar programas FORTRAN mal projetados em programas modernos padrão FORTRAN ou C.

Links para uma variedade de ferramentas de engenharia e engenharia reversa podem ser encontrados em www.csse.monash.edu/~ipeake/reeng/free-swre-tools.html e www.cs.ualberta.ca/~kenw/toolsdir/all.html.

29.8 ENGENHARIA DIRETA

? Que opções existem quando encontramos um programa mal projetado e mal implementado?

Um programa com um fluxo de controle, que graficamente equivale a um emaranhado, com módulos de 2 mil instruções, algumas poucas linhas de comentários em 290 mil instruções de código-fonte e nenhuma outra documentação deve ser modificado para acomodar alterações e requisitos de usuário. Temos as seguintes opções:

1. Podemos trabalhar arduamente fazendo modificação após modificação, enfrentando os problemas do projeto *ad hoc* e do código-fonte confuso para implementar as mudanças necessárias.
2. Podemos tentar entender os detalhes internos do programa em um esforço para tornar as modificações mais eficazes.
3. Podemos reprojeter, recodificar e testar as partes do software que requerem modificação, aplicando uma abordagem de engenharia de software a todos os segmentos revisados.
4. Podemos reprojeter completamente, recodificar e testar o programa, usando ferramentas de reengenharia para ajudar a entendermos o projeto atual.



A reengenharia é bem semelhante à limpeza de seus dentes. Você pode pensar em mil razões para adiar, e conseguirá um jeito de retardar por um tempo. Mas, eventualmente, sua tática de adiamento se voltará contra você, causando-lhe dor.

Não há uma única opção “correta”. As circunstâncias podem recomendar a primeira opção mesmo que as outras sejam mais desejáveis.

Em vez de esperar até que seja solicitada uma manutenção, a organização de desenvolvimento ou suporte usa os resultados da análise de inventário para selecionar um programa que (1) permanecerá em uso por certo número de anos, (2) está sendo utilizado no momento com sucesso e (3) pode passar por modificações importantes ou melhoramentos em futuro próximo. Então é aplicada a opção 2, 3 ou 4.

Em uma primeira impressão, a sugestão para que você redesenvolva um grande programa quando uma versão ainda funcione pode ser bastante extravagante. Mas antes de fazer um julgamento, considere os seguintes pontos:

⁹ As ferramentas aqui apresentadas não significam um aval, mas sim uma amostra dessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

1. O custo para manter uma linha de código-fonte pode ser 20 a 40 vezes o do desenvolvimento inicial daquela linha.
2. O reprojeto da arquitetura de software (programa e/ou estrutura de dados), usando conceitos modernos de projeto, pode facilitar muito a manutenção futura.
3. Como já existe um protótipo do software, a produtividade do desenvolvimento deverá ser muito mais alta do que a média.
4. O usuário agora tem experiência com o software. Portanto, novos requisitos e a direção das mudanças podem ser definidos com grande facilidade.
5. Ferramentas automatizadas para reengenharia facilitarão algumas partes do trabalho.
6. Existirá uma configuração de software completa (documentos, programas e dados) depois de uma manutenção preventiva.

Um grande departamento de desenvolvimento de software interno (por exemplo, um grupo de desenvolvimento de sistemas comerciais para uma grande empresa de produtos de consumo) pode ter de 500 a 2 mil programas em produção sob sua responsabilidade. Esses programas podem ser classificados por importância e examinados como candidatos para engenharia direta.

O processo de engenharia direta aplica os princípios, conceitos e métodos de engenharia de software, para recriar um aplicativo. Em muitos casos, a engenharia direta não cria só um equivalente moderno de um programa antigo. Em vez disso, novos requisitos de usuário e tecnologia são integrados ao trabalho de reengenharia. O programa redesenvolvido amplia a capacidade do aplicativo antigo.

29.8.1 Engenharia direta para arquiteturas cliente-servidor

Durante as últimas décadas, muitos aplicativos para mainframes passaram por operações de reengenharia para acomodar arquiteturas cliente-servidor (incluindo WebApps). Essencialmente, recursos de computação centralizados (incluindo software) são distribuídos entre muitas plataformas cliente. Embora possa ser projetada uma variedade de diferentes ambientes distribuídos, a aplicação para mainframes típica que passa por uma reengenharia, transformando-se em uma arquitetura cliente-servidor, tem as seguintes características:

- A funcionalidade da aplicação migra para cada computador cliente.
- Novas interfaces GUI são implementadas nas instalações do cliente.
- Funções de base de dados são atribuídas ao servidor.
- Funcionalidade especializada (por exemplo, análise que demanda muitos cálculos) podem permanecer no servidor.
- Novos requisitos de comunicação, segurança, arquivamento e controle podem ser estabelecidos tanto no lado do cliente quanto no do servidor.



Em alguns casos, a migração para uma arquitetura cliente-servidor deverá ser considerada não como uma reengenharia, mas como um novo trabalho de desenvolvimento. A reengenharia entra em cena apenas quando funcionalidade específica do sistema antigo deve ser integrada em uma nova arquitetura.

É importante notar que a migração de computação mainframe para cliente-servidor requer reengenharia tanto de negócio quanto de software. Além disso, deverá ser estabelecida uma “infraestrutura de rede corporativa” [Jay94].

A reengenharia para aplicações cliente-servidor começa com uma análise completa do ambiente comercial que abrange o mainframe existente. Três leis de abstração podem ser identificadas. A *base de dados* estabelece a fundação de uma arquitetura cliente-servidor e gerencia transações e consultas dos aplicativos do servidor. No entanto, essas transações e consultas devem ser controladas com base em um conjunto de regras de negócio (definidas por um processo de negócio existente ou que passou por uma reengenharia). Aplicações cliente fornecem funcionalidade destinada à comunidade de usuários.

As funções do sistema de gerenciamento de base de dados e a arquitetura de dados da base de dados existentes devem passar por uma engenharia reversa como precursoras para o reprojeto da camada fundamental da base de dados. Em alguns casos, é criado um modelo de dados

(Capítulo 6). De qualquer forma, a base de dados cliente-servidor passa por uma reengenharia para garantir que as transações sejam executadas de uma maneira consistente, que todas as atualizações sejam executadas somente por usuários autorizados, que as regras de negócio básicas sejam seguidas (por exemplo, antes que o registro de um fornecedor seja deletado, o servidor assegura que não existem contas a pagar, contratos, ou comunicações para aquele fornecedor), que as consultas possam ser acomodadas eficientemente e que seja estabelecida uma capacidade total de arquivamento.

A camada das regras de negócio representa software residente tanto no cliente quanto no servidor. Esse software executa tarefas de controle e coordenação para assegurar que as transações e consultas entre o aplicativo cliente e a base de dados estejam em conformidade com processos de negócio estabelecidos.

A camada dos aplicativos cliente implementa funções de negócio requeridas por grupos específicos de usuários finais. Em muitas ocasiões, um aplicativo para mainframe é segmentado em uma série de aplicativos para desktop menores e modificados por meio da reengenharia. A comunicação entre os aplicativos para desktop (quando necessária) é controlada pela camada de regras de negócio.

Uma discussão ampla sobre projeto e reengenharia de software cliente-servidor pode ser encontrada em livros dedicados ao assunto. Se você tiver interesse, veja [Van02], [Cou00], ou [Orf99].

29.8.2 Engenharia direta para arquiteturas orientadas a objeto

A reengenharia de software orientada a objeto tornou-se o paradigma de desenvolvimento preferido por muitas organizações. Mas o que dizer sobre os aplicativos desenvolvidos usando-se métodos convencionais? Em alguns casos, a resposta é deixar esses aplicativos “como estão”. Em outros, mais antigos, devem passar por uma reengenharia para que possam ser facilmente integrados em sistemas orientados a objeto de grande porte.

A reengenharia de software convencional para uma implementação orientada a objeto usa muitas das mesmas técnicas discutidas na Parte 2 deste livro. Primeiro, o software passa por uma engenharia reversa de modo que possam ser criados modelos de dados, funcional e comportamental apropriados. Se o sistema ao qual se aplica a reengenharia amplia a funcionalidade ou comportamento do aplicativo original, são criados casos de uso (Capítulos 5 e 6). Os modelos de dados desenvolvidos durante a engenharia reversa são então utilizados em conjunto com a modelagem CRC (Capítulo 6) para estabelecer a base para a definição de classes. Hierarquias de classe, modelos de relacionamenmto de objetos, modelos de comportamento de objeto e sub-sistemas são definidos, e inicia-se o projeto orientado a objeto.

Na medida em que a engenharia direta orientada a objeto avança da análise para o projeto, pode ser invocado um modelo de processo CBSE (Capítulo 10). Se o aplicativo existente reside dentro de um domínio que já seja constituído por muitos aplicativos orientados a objeto, é provável que exista uma biblioteca de componentes robusta e que possa ser empregada durante a engenharia direta.

Para aquelas classes que devem ser criadas desde o início, pode ser possível reutilizar algoritmos e estruturas de dados do aplicativo convencional existente. No entanto, estes devem ser reprojitados para ficar em conformidade com a estrutura orientada a objeto.

29.9 A ECONOMIA DA REENGENHARIA

Em um mundo perfeito, todo programa não manutenível seria aposentado imediatamente, para ser substituído por aplicativos de alta qualidade, desenvolvidos usando modernas práticas de engenharia de software. Mas vivemos em um mundo de recursos limitados. A reengenharia consome recursos que podem ser utilizados para outras finalidades de negócio. Portanto, antes de pensar em fazer a reengenharia de um aplicativo existente, deve-se executar uma análise de custo-benefício.

“Você pode gastar um pouco agora ou gastar muito mais tarde.”

Cartaz em uma revendedora de automóveis sugerindo manutenção no carro

Um modelo de análise custo-benefício para reengenharia foi proposto por Sneed [Sne95]. São definidos nove parâmetros:

- P_1 = custo anual corrente de manutenção para um aplicativo
- P_2 = custo anual corrente das operações para um aplicativo
- P_3 = valor de negócio anual corrente de um aplicativo
- P_4 = custo de manutenção anual previsto após a reengenharia
- P_5 = custo anual previsto das operações após a reengenharia
- P_6 = valor de negócio anual previsto após a reengenharia
- P_7 = custos estimados da reengenharia
- P_8 = prazo estimado para fazer a reengenharia
- P_9 = fator de risco da reengenharia ($P_9 = 1,0$ é nominal)
- L = expectativa de vida do sistema

O custo associado com a manutenção continuada de um aplicativo candidato à reengenharia (a reengenharia ainda não foi feita) pode ser definido como

$$C_{\text{manut}} = [P_3 - (P_1 + P_2)] \times L \quad (29.1)$$

Os custos associados com a reengenharia são definidos usando a seguinte relação:

$$C_{\text{reeng}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \quad (29.2)$$

Usando os custos apresentados nas Equações (29.1) e (29.2), o benefício total da reengenharia pode ser calculado como

$$\text{Custo-benefício} = C_{\text{reeng}} - C_{\text{manut}} \quad (29.3)$$

A análise custo-benefício apresentada nessas equações pode ser feita para todos os aplicativos de alta prioridade identificados durante a análise de inventário (Seção 29.4.2). Aqueles aplicativos que apresentam o custo-benefício mais alto podem ser identificados para reengenharia, enquanto o trabalho com os outros aplicativos pode ser adiado até que os recursos estejam disponíveis.

29.10 RESUMO

A manutenção e o suporte de software são atividades contínuas que ocorrem por todo o ciclo de vida de um aplicativo. Durante essas atividades, defeitos são corrigidos, aplicativos são adaptados a um ambiente operacional ou de negócio em mutação, melhorias são implementadas por solicitação dos interessados e é fornecido suporte aos usuários quando integram um aplicativo em seu fluxo de trabalho pessoal ou corporativo.

A reengenharia ocorre em dois níveis de abstração. No nível de negócio, ela concentra-se nos processos de negócio para melhorar a competitividade em alguma área do negócio. No nível de software, a reengenharia examina os sistemas de informação e os aplicativos, com a finalidade de reestruturá-los para que tenham uma melhor qualidade.

A reengenharia de processos de negócio (*business process reengineering* - BPR) define metas de negócio; identifica e avalia processos de negócio existentes (no contexto das metas definidas); especifica e projeta processos revisados; cria protótipos, refina e os viabiliza em um negócio. A BPR tem um foco que se estende além do software. O resultado da BPR é muitas vezes a definição de maneiras pelas quais a tecnologia de informação pode suportar melhor o negócio.

A reengenharia de software abrange uma série de atividades que incluem análise de inventário, reestruturação da documentação, engenharia reversa, reestruturação de programas e dados e engenharia direta. A finalidade dessas atividades é criar versões dos programas existentes que tenham melhor qualidade e melhor manutenibilidade — programas que serão viáveis para o século 21.

O custo-benefício da reengenharia pode ser determinado quantitativamente. O custo do estado atual, isto é, o custo associado ao suporte e manutenção continuados de um aplicativo existente, é comparado aos custos projetados da reengenharia e a redução resultante nos custos de manutenção e suporte. Em quase todos os casos nos quais um programa tem uma vida longa e apresenta no momento uma manutenibilidade ou suportabilidade ruim, a reengenharia representa uma estratégia de negócio eficiente em termos de custo.

PROBLEMAS E PONTOS A PONDERAR

- 29.1.** Considere um trabalho que tenha realizado nos últimos cinco anos. Descreva os processos de negócio nos quais você tomou parte. Use o modelo BPR descrito na Seção 29.4.2 para recomendar alterações no processo em um esforço para torná-lo mais eficiente.
- 29.2.** Faça alguma pesquisa sobre a eficiência da reengenharia dos processos de negócio. Apresente argumentos a favor e contra essa abordagem.
- 29.3.** O seu professor selecionará um dos programas que todos na classe tenham desenvolvido durante o curso. Troque o seu programa de forma aleatória com algum outro aluno na classe. Não explique nem detalhe o programa. Agora, implemente uma melhoria (especificada pelo seu professor) no programa que você recebeu.
- Execute todas as tarefas de engenharia de software incluindo um breve detalhamento (mas não com o autor do programa).
 - Mantenha um controle preciso de todos os erros encontrados durante o teste.
 - Discuta sua experiência na classe.
- 29.4.** Explore a checklist da análise de inventário apresentada no site da SEPA e tente desenvolver um sistema de classificação quantitativa de software que poderia ser aplicado a programas existentes, em um trabalho para escolher programas candidatos para a reengenharia. O seu sistema deve se estender além da análise econômica apresentada na Seção 29.9.
- 29.5.** Sugira alternativas em documentação impressa ou eletrônica que poderia servir de base para a reestruturação da documentação. (Sugestão: pense em novas tecnologias descritivas que poderiam ser usadas para mostrar a finalidade do software.)
- 29.6.** Algumas pessoas acreditam que a tecnologia de inteligência artificial aumentará o nível de abstração do processo de engenharia reversa. Faça alguma pesquisa sobre esse assunto (isto é, o uso de inteligência artificial (*artificial intelligence* - AI) para a engenharia reversa) e escreva um pequeno artigo que destaca esse ponto.
- 29.7.** Por que a completeza é difícil de ser atingida na medida em que o nível de abstração aumenta?
- 29.8.** Por que a interatividade deve aumentar se a completeza tiver de aumentar?
- 29.9.** Usando informações obtidas na Web, apresente características de três ferramentas de engenharia reversa para a sua classe.
- 29.10.** Há uma diferença sutil entre reestruturação e engenharia direta. Qual é?
- 29.11.** Pesquise a literatura e/ou fontes na Internet para encontrar um ou mais artigos que discutem estudos de casos de reengenharia de mainframe para cliente-servidor. Apresente um resumo.
- 29.12.** Como você determinaria P_4 até P_7 no modelo de custo-benefício apresentado na Seção 29.9?

LEITURAS E FONTE DE INFORMAÇÃO COMPLEMENTARES

É irônico que a manutenção e suporte de software representem as atividades mais custosas na vida útil de um aplicativo e, no entanto, foram escritos menos livros sobre manutenção e suporte do que sobre qualquer outro tópico importante de engenharia de software. Entre as recentes adições à literatura estão os livros de Jarzabek (*Effective Software Maintenance and Evolution*, Auerbach, 2007), Grubb e Takang (*Software Maintenance: Concepts and Practice*, World

Scientific Publishing Co., 2d ed., 2003), e Pigoski (*Practical Software Maintenance*, Wiley, 1996). Esses livros abrangem as práticas básicas de manutenção e suporte e apresentam diretrizes úteis de gerenciamento. As técnicas de manutenção que focalizam ambientes cliente-servidor são discutidas por Schneberger (*Client/Server Software Maintenance*, McGraw-Hill, 1997). A pesquisa atual em “evolução de software” é apresentada em uma antologia editada por Mens e Demeyer (*Software Evolution*, Springer, 2008).

Como muitos tópicos atuais na área de negócios, a euforia que envolve a reengenharia de processos de negócio deu origem a uma visão mais pragmática do assunto. Hammer e Champy (*Reengineering the Corporation*, HarperBusiness, revised edition, 2003) fomentaram logo o interesse com sua obra que se tornou best-seller. Outros livros de Smith e Fingar [*Business Process Management (BPM): The Third Wave*, Meghan-Kiffer Press, 2003], Jacka e Keller (*Business Process Mapping: Improving Customer Satisfaction*, Wiley, 2001), Sharp e McDermott (*Workflow Modeling*, Artech House, 2001), Andersen (*Business Process Improvement Toolbox*, American Society for Quality, 1999), e Harrington et al. (*Business Process Improvement Workbook*, McGraw-Hill, 1997) apresentam estudos de casos e diretrizes detalhadas para BPR.

Fong (*Information Systems Reengineering and Integration*, Springer, 2006) descreve técnicas de conversão de base de dados, engenharia reversa e engenharia direta da forma como são aplicadas para a maioria dos sistemas de informação. Demeyer e seus colegas (*Object Oriented Reengineering Patterns*, Morgan Kaufmann, 2002) proporcionam uma visão baseada em padrões sobre como refazer e/ou recriar sistemas orientados a objeto. Secord e seus colegas (*Modernizing Legacy Systems*, Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies*, Prentice Hall, 2002), Valenti (*Successful Software Reengineering*, IRM Press, 2002), e Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software*, Fitzroy Dearborn Publishers, 1999) concentram-se em estratégias e práticas para reengenharia em nível técnico. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) “proporciona uma estrutura para manter sistemas de aplicativos sincronizados com as estratégias de negócio e com as mudanças da tecnologia”.

Cameron (*Reengineering Business for Success in the Internet Age*, Computer Technology Research, 2000) e Umar [*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*, Prentice Hall, 1997] fornecem orientações para organizações que querem transformar sistemas legados em um ambiente baseado na Web. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall, 1996) aborda a ligação entre BPR e tecnologia da informação. Aiken (*Data Reverse Engineering*, McGraw-Hill, 1996) discute como reunir, reorganizar e reutilizar dados organizacionais. Arnold (*Software Reengineering*, IEEE Computer Society Press, 1993) montou uma excelente antologia dos primeiros artigos que focalizavam as tecnologias de reengenharia de software.

Uma grande variedade de fontes de informação sobre reengenharia de software está disponível na Internet. Uma lista atualizada das referências da Web relevantes para a manutenção e reengenharia de software pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

TÓPICOS AVANÇADOS

Nesta última parte do livro, consideraremos uma série de temas avançados que ampliarão seu entendimento sobre a engenharia de software. Nos próximos capítulos serão tratadas as seguintes questões:

- O que é a melhoria do processo de software e como pode ser usada para melhorar as práticas de engenharia de software?
- Quais as tendências emergentes que podem ter uma influência significativa sobre as práticas de engenharia de software na próxima década?
- Qual é a expectativa para os engenheiros de software?

Uma vez respondidas essas perguntas, você entenderá de assuntos que podem ter um profundo impacto sobre a engenharia de software nos próximos anos.

**CONCEITOS-
-CHAVE**

avaliação	686
avaliação	689
CMMI	691
CMM das pessoas ..	695
educação e treinamento	687
fatores críticos de sucesso	690
gerenciamento de riscos	689
instalação/ migração	689
justificação	688
modelos de maturidade	685
retorno sobre o investimento	697
seleção	688
melhora do processo de software (SPI) ..	685
aplicabilidade	686
estruturas	683
processo	686

Muito antes de estar na moda, a frase “melhoria do processo de software”, trabalhei em grandes corporações tentando melhorar a qualidade de suas práticas de engenharia de software. Como resultado de minhas experiências, escrevi o livro *Making software engineering happen*. O prefácio apresenta o seguinte comentário:

Durante os últimos dez anos tive a oportunidade de ajudar muitas grandes empresas a implementar práticas de engenharia de software. A tarefa é difícil e raramente ocorre tão tranquilamente quanto se gostaria — mas quando é bem-sucedida, os resultados são profundos. Os projetos de software têm maior possibilidade de se completarem a tempo. As comunicações entre todas as partes envolvidas no desenvolvimento de software são melhoradas. O nível de confusão e caos que com frequência prevalece para grandes projetos de software é reduzido substancialmente. O número de erros encontrados pelo cliente diminui substancialmente. A credibilidade da organização de software melhora. E a gerência tem um problema a menos com que se preocupar.

Mas nem tudo é um mar de rosas. Muitas empresas tentam implementar práticas de engenharia de software e desistem frustradas. Outras executam tudo pela metade e nunca verão os benefícios que acabamos de descrever. Há ainda outras que o fazem da forma linha linha-dura, impositiva, o que resulta em rebelião declarada entre o pessoal técnico e os gerentes e subsequente perda de moral.

Embora essas palavras tenham sido escritas há mais de 20 anos, permanecem ainda válidas nos dias atuais.

Enquanto nos aproximamos da segunda década do século 21, muitas grandes organizações têm tentado “fazer acontecer a engenharia de software”. Algumas implementaram práticas individuais que ajudaram a melhorar a qualidade do produto e o prazo de entrega. Outras estabeleceram um processo de software “consolidado” que orienta suas atividades

PANORAMA

O que é? A melhoria da gestão de qualidade abrange uma série de atividades que levarão a um melhor processo de software e, em consequência, melhor qualidade do software fornecido com melhor prazo de entrega.

Quem realiza? As pessoas que aderem à melhoria da gestão de qualidade (*software process improvement – SPI*) são provenientes de três grupos: gerentes técnicos, engenheiros de software e indivíduos com responsabilidade pela garantia da qualidade.

Por que é importante? Algumas organizações de software têm pouco mais do que um processo de software específico. À medida que trabalham para melhorar suas práticas, devem resolver os pontos fracos do processo e tentar melhorar sua abordagem para o trabalho de software.

Quais são as etapas envolvidas? A abordagem para a SPI é iterativa e contínua, mas pode ser encarada em cinco etapas: (1) avaliação do processo de software atual, (2) educação e treinamento dos profissionais e gerentes, (3) seleção e justificação de elementos do processo, métodos de engenharia de software e ferramentas, (4) implementação do plano da SPI e (5) avaliação e ajuste com base nos resultados do plano.

Qual é o artefato? Embora haja muitos produtos SPI intermediários, o resultado final é um processo de software melhorado que leva a uma qualidade mais alta do software.

Como garantir que o trabalho foi realizado corretamente? O software que a sua organização produz será fornecido com menos defeitos, o retrabalho em cada estágio do processo de software será reduzido e a entrega no prazo se tornará muito mais possível.

técnicas e de gerenciamento de projeto. Mas há aquelas que continuam a lutar com dificuldades. Suas práticas são às vezes boas e outras não, e o processo é improvisado. Ocasionalmente, o trabalho é espetacular, mas, na maior parte, cada projeto é uma aventura, e ninguém sabe se terminará bem ou mal.

Assim, qual desses dois grupos precisa de melhoria do processo de software? A resposta (que pode surpreendê-lo) é *ambos*. Os que tiveram sucesso ao fazer a engenharia de software acontecer não podem se tornar complacentes. Devem trabalhar continuamente para melhorar sua abordagem de engenharia de software. E os que lutam com dificuldades devem começar a tomar seus rumos em direção às melhorias.

30.1 O QUE É SPI?

PONTO-CHAVE

SPI implica um processo de software definido, uma abordagem organizacional e estratégia para melhoria.

“Grande parte da crise de software é autoimposta, segundo um CIO, ‘Prefiro ter o software errado e não atrasado. Sempre podemos corrigir depois.’”

Mark Paulk

O termo *melhoria do processo de software* (*software process improvement* - SPI) envolve muitas coisas. Primeiro, implica que os elementos de um processo de software eficaz podem ser definidos de maneira eficaz; segundo, que uma abordagem organizacional existente para o desenvolvimento de software pode ser avaliada em relação aos elementos; e terceiro, que uma estratégia significativa para melhoria pode ser definida. A estratégia SPI transforma a abordagem existente para o desenvolvimento de software em algo mais focado, com melhor repetibilidade e mais confiável (em termos de qualidade de produto e prazo de entrega).

Como a SPI não é gratuita, deve produzir um retorno do investimento. O trabalho e o tempo necessário para implementar uma estratégia SPI deve se pagar de alguma maneira mensurável. Para tanto, os resultados do processo e prática melhorados devem levar a uma redução nos “problemas” de software que custam tempo e dinheiro. Eles devem reduzir: o número de defeitos que passam para os usuários finais, o volume de retrabalho causado por problemas de qualidade, os custos associados à manutenção e suporte do software (Capítulo 29) e os custos indiretos que ocorrem quando o software é entregue com atraso.

30.1.1 Abordagens para SPI

Embora a organização possa decidir por uma abordagem relativamente informal para SPI, a grande maioria escolhe dentre uma série de estruturas SPI. A *estrutura SPI* define (1) inúmeras características que devem estar presentes quando se deseja obter um processo de software eficaz, (2) um método para determinar se aquelas características estão presentes, (3) um mecanismo para resumir os resultados de qualquer avaliação, e (4) uma estratégia para ajudar a organização na implementação das características de processo consideradas fracas ou ausentes.

A estrutura SPI avalia a “maturidade” do processo de software de uma organização e fornece indicação qualitativa de um nível de maturidade; com frequência, aplica-se o termo “modelo de maturidade” (Seção 30.1.2). Essencialmente, a estrutura SPI abrange um modelo de maturidade que, por sua vez, incorpora uma série de indicadores de qualidade de processo que fornecem indicação geral da qualidade do processo que levará à qualidade do produto.

A Figura 30.1 fornece uma visão geral de uma estrutura SPI típica. São mostrados os elementos-chave da estrutura e suas relações uns com os outros.

Você deve observar que não há uma estrutura SPI universal. De fato, a estrutura SPI escolhida por uma organização reflete o grupo de pessoas que está apoiando o trabalho da SPI. Conradi [Con96] define seis diferentes grupos de suporte a SPI:

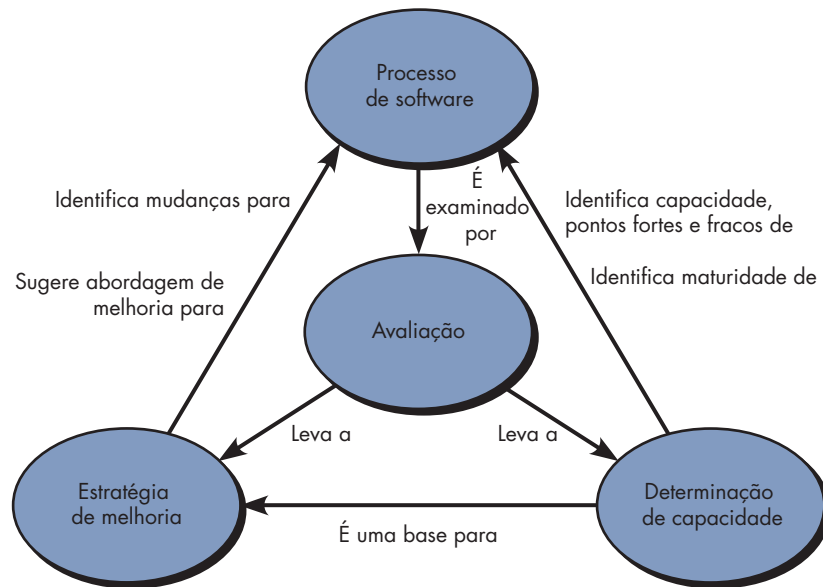
Certificadores de qualidade. Os esforços para melhoria de processo apoiados por esse grupo concentram-se na seguinte relação:

$$\text{Qualidade(Processo)} = \text{Qualidade(Produto)}$$

“Quais são os grupos que apoiam um trabalho SPI?”

FIGURA 30.1**Elementos de uma estrutura SPI**

Fonte: Adaptado de (Rou02)



A abordagem desse grupo enfatiza métodos de avaliação e examina uma série bem definida de características que lhes permitem determinar se o processo apresenta qualidade. São mais propensos a adotar uma estrutura de processo do tipo CMM, SPICE, TickIT ou Bootstrap.¹

Formalistas. Esse grupo quer entender (e, quando possível, otimizar) o fluxo de trabalho de processo. Para tanto, usam linguagens de modelagem de processo (*process modeling languages* - PMLs) para criar um modelo do processo existente e então projetar extensões ou modificações que tornarão o processo mais eficaz.

Defensores das ferramentas. Esse grupo insiste em uma abordagem assistida por ferramenta para SPI, que modela o fluxo de trabalho e outras características de processo, de maneira a ser analisada para melhoria.

Profissionais. Esse grupo usa uma abordagem pragmática, “enfazando gerenciamento básico de projeto, qualidade e produto, aplicando planejamento e métricas em nível de projeto, mas com pouca modelagem formal de processo ou suporte” [Con96].

Reformadores. O objetivo desse grupo é a mudança organizacional que pode levar a um melhor processo de software. Eles tendem a concentrarem-se mais nos aspectos humanos (Seção 30.5) e enfatizam medidas de capacidade humana e estrutura.

Ideologistas. Esse grupo focaliza a adequação de um modelo particular de processo para um domínio de aplicação específico ou estrutura organizacional. Em vez de modelos de processo de software típicos (por exemplo, modelos iterativos), os ideologistas teriam um interesse maior em um processo que poderia, digamos, suportar reutilização ou reengenharia.

Quando uma estrutura SPI é aplicada, os grupos que a patrocinam (independentemente de seu foco geral) devem estabelecer mecanismos para: (1) suportar transição de tecnologia, (2) determinar o grau segundo o qual uma organização está pronta para absorver mudanças de processo propostas, e (3) medir o grau segundo o qual as mudanças foram adotadas.

¹ Cada uma dessas estruturas SPI será discutida mais adiante neste capítulo.

? “Quais são os diferentes grupos que suportam SPI?”

30.1.2 Modelos de maturidade

Um *modelo de maturidade* é aplicado no contexto de uma estrutura SPI. Sua finalidade é proporcionar uma indicação geral da “maturidade do processo” exibida por uma organização de software. Por exemplo, a indicação da qualidade do processo de software, o grau segundo o qual os profissionais entendem e aplicam o processo, e o estado geral da prática de engenharia de software. Para tanto, utiliza-se algum tipo de escala ordinal.

Por exemplo, o *Capability Maturity Model* (Seção 30.4) do Software Engineering Institute sugere cinco níveis de maturidade [Sch96]:

Nível 5, Otimizado – A organização tem sistemas quantitativos de realimentação para identificar os pontos fracos do sistema para corrigi-los proativamente. As equipes de projeto analisam os defeitos para determinar suas causas; os processos de software são avaliados e atualizados para evitar que defeitos conhecidos voltem a ocorrer.

Nível 4, Controlado – Métricas detalhadas de processo de software e qualidade de produto estabelecem os fundamentos de avaliação quantitativa. As variações significativas no desempenho do processo podem ser distinguidas do ruído aleatório e previstas as tendências no processo e na qualidade do produto.

Nível 3, Definido – Processos para gerenciamento e engenharia são documentados, padronizados e integrados em um processo de software padrão para a organização. Todos os projetos usam uma versão aprovada, personalizada do processo de software padrão da organização para desenvolvimento de software.

Nível 2, Reproduzível – São estabelecidos processos básicos de gerenciamento de projeto para rastrear o custo, cronograma e funcionalidade. O planejamento e gerenciamento de novos produtos baseiam-se na experiência com projetos similares.

Nível 1, Inicial – Poucos processos são definidos, e o sucesso depende mais dos esforços heroicos individuais do que seguir um processo e usar um trabalho de equipe em sinergia.

A escala de maturidade CMM não vai além, mas a experiência indica que muitas organizações apresentam níveis de “imaturidade de processo” [Sch96] que enfraquece qualquer tentativa racional de melhorar as práticas de engenharia de software. Schorsch [Sch06] sugere quatro níveis de imaturidade muitas vezes encontrados nas organizações de desenvolvimento de software:

Nível 0, Negligente – Quando não se permite o processo de desenvolvimento bem-sucedido. Todos os problemas são considerados técnicos. As atividades gerenciais e de garantia de qualidade são consideradas complicadas e supérfluas para a tarefa do processo de desenvolvimento de software. Dependem de amuletos.

Nível – 1, Obstrutivo – Impõem-se processos contraproduativos. Os processos são rigidamente definidos e exagera-se na sua obediência. Rituais cerimoniosos ocorrem em abundância. O gerenciamento coletivo inviabiliza a atribuição de responsabilidade. *Status quo über alles* (Por cima de todos).

Nível – 2, Arrogante – Desprezo completo pela boa engenharia de software. Uma separação total entre atividades de desenvolvimento de software e atividades de melhora do processo de software. Ausência completa de um programa de treinamento.

Nível – 3, Sabotador – Negligência total do próprio compromisso, descrédito consciente dos esforços de melhoria do processo de software da organização. Falta de incentivo e mau desempenho.

Os níveis de imaturidade de Schorsch são prejudiciais para qualquer organização de software. Se você encontrar qualquer um deles, as tentativas de SPI estão fadadas ao fracasso.

A questão fundamental é se as escalas de maturidade, como a proposta como parte da CMM, proporcionam qualquer benefício real. Penso que sim. A escala de maturidade proporciona uma visão clara da qualidade do processo que pode ser utilizado pelos profissionais e gerentes, como um benchmark a partir do qual podem ser planejadas as estratégias de melhorias.

30.1.3 A SPI é para todos?

Por muitos anos, a SPI foi vista como uma atividade “corporativa” – eufemismo para algo que apenas grandes empresas executam. Mas nos dias atuais, uma porcentagem significativa de todo o desenvolvimento de software está sendo executada por empresas que empregam menos

PONTO-CHAVE

O modelo de maturidade define níveis de competência no processo de software e implementação.

“Como você reconhece uma organização que resistirá aos esforços de SPI?”



Se um modelo de processo específico ou uma abordagem SPI parecem ser opressivos para sua organização, provavelmente são.

de cem pessoas. Uma organização pequena pode empreender atividades de SPI e executá-las de maneira bem-sucedida?

Há diferenças culturais importantes entre organizações de desenvolvimento de software grandes e pequenas. Não é surpresa o fato de que pequenas organizações são mais informais, aplicam menos práticas padronizadas e tendem a ser auto-organizadas. Elas também tendem a se vangloriar da “criatividade” de membros da organização de software e encaram, inicialmente, uma estrutura SPI como demasiadamente burocrática e de grande peso. No entanto, a melhoria do processo é tão importante para a organização pequena quanto para a grande.

Em pequenas organizações, a implementação de uma estrutura SPI requer recursos que podem ser escassos. Os gerentes precisam alocar pessoas e dinheiro para fazer acontecer a engenharia de software. Portanto, independentemente do tamanho da organização de software, é razoável considerar a motivação comercial para a SPI.

A SPI será aprovada e implementada somente depois que seus proponentes demonstrarem uma alavancagem financeira [Bir98]. A alavancagem financeira é demonstrada examinando-se os benefícios técnicos (por exemplo, menos defeitos que passam para o campo, retrabalho reduzido, menores custos de manutenção, ou uma chegada ao mercado mais rápida) e traduzindo-os em dólares. Essencialmente, você deve mostrar um retorno realístico sobre o investimento (Seção 30.7) para os custos SPI.

30.2 O PROCESSO DE SPI

A parte difícil da SPI não é a definição de características que caracterizam um processo de software de alta qualidade ou a criação de um modelo de maturidade de processo. Essas coisas são relativamente fáceis. No entanto, a parte difícil é estabelecer um consenso para iniciar a SPI e definir uma estratégia contínua para implementá-la por meio de uma organização de software.

O Software Engineering Institute desenvolveu a IDEAL – “modelo de melhoria organizacional que serve de roteiro para iniciar, planejar e implementar ações de melhoria” [SEI08]. A IDEAL representa os muitos modelos de processo para SPI, definindo cinco atividades distintas – iniciar, diagnosticar, estabelecer, agir e aprender – que orientam uma organização com base nas atividades de SPI.

Neste livro, apresento um roteiro um pouco diferente para a SPI, baseado no modelo de processo para SPI proposto originalmente em [Pre88]. Ele aplica uma filosofia de bom senso que requer organização para que a empresa (1) se autoavalie, (2) depois se torne mais inteligente para que possa fazer escolhas inteligentes, (3) selecione o modelo de processo (e os elementos de tecnologia relacionados) que melhor satisfaça às suas necessidades, (4) crie uma instância do modelo em seu ambiente operacional e sua cultura, e (5) avalie o que foi feito. Essas cinco atividades (discutidas nas próximas subseções)² são aplicadas de maneira iterativa (cíclica) para fortalecer a melhoria contínua do processo.

30.2.1 Avaliação e análise de lacunas

Qualquer tentativa de melhorar o processo de software sem primeiro avaliar a eficácia das atividades da estrutura e práticas de engenharia de software associadas seria o mesmo que iniciar uma longa jornada a uma localidade sem nenhuma ideia de onde se deve partir. Você iniciará com grande entusiasmo, vai tentar encontrar seu rumo, vai gastar muita energia e suportar grandes doses de frustração e, provavelmente, decidirá que realmente quer ir a lugar nenhum. Resumindo, antes de começar sua jornada, é aconselhável saber precisamente onde você está.

² Parte do conteúdo dessas seções foi adaptado de [Pre88] com permissão.



Procure entender seus pontos fortes e fracos. Se for inteligente, trabalhará sobre os pontos fortes.

A primeira atividade do roteiro, chamada de *avaliação*, permite que você tome o seu rumo. A finalidade da avaliação é revelar os pontos fortes e fracos na maneira como sua organização aplica processos de software existentes e as práticas de engenharia de software que compõem o processo.

A avaliação examina uma grande variedade de ações e tarefas que levarão a um processo de alta qualidade. Por exemplo, independentemente do modelo de processo escolhido, a organização deve estabelecer mecanismos genéricos como: abordagens definidas para comunicação com o cliente; métodos estabelecidos para representar os requisitos do usuário; definição de uma estrutura de gerenciamento de projeto que inclui definição de escopo, estimativa, cronograma e rastreamento de projeto; métodos de análise de risco; procedimentos de gerenciamento de alterações; garantia de qualidade e atividades de controle, incluindo revisões; e muitas outras. Cada um desses mecanismos é considerado no contexto das atividades de estrutura e proteção (Capítulo 2) estabelecido e avaliado para determinar se uma das seguintes questões pode ser resolvida:

- O objetivo da ação está claramente definido?
- Os produtos requeridos como entrada e produzidos como saída estão identificados e descritos?
- Os trabalhos a ser realizados estão claramente descritos?
- As pessoas que devem executar as ações estão identificadas de acordo com suas funções?
- Os critérios de entrada e saída foram estabelecidos?
- As métricas para a ação foram estabelecidas?
- Há ferramentas disponíveis para suportar a ação?
- Há um programa de treinamento explícito que trata da ação?
- A ação é executada uniformemente para todos os projetos?

Embora as questões apresentadas impliquem uma resposta *sim* ou *não*, o papel da avaliação é examinar a resposta para determinar se a ação em questão está sendo executada conforme as melhores práticas.

À medida que é conduzida a avaliação do processo, você (ou aqueles que foram contratados para executar a avaliação) deverá também concentrar-se nos seguintes atributos:



“Que atributos genéricos você procura durante a avaliação?”

Consistência. As atividades importantes, ações e tarefas são aplicadas de forma consistente em todos os projetos de software e por todas as equipes de software?

Sofisticação. As ações de gerência e técnicas são executadas com um nível de sofisticação que envolve completo entendimento das melhores práticas?

Aceitação. O processo de software e as práticas de engenharia de software são amplamente aceitos pela gerência e pessoal técnico?

Compromisso. A gerência forneceu os recursos necessários para obter consistência, sofisticação e aceitação?

A diferença entre aplicação local e melhor prática representa um “vazio” que oferece oportunidades para melhoria. O grau segundo o qual a consistência, sofisticação, aceitação e compromisso são atingidos indica o nível de mudança cultural necessário para atingir uma melhora significativa.

30.2.2 Educação e treinamento

Embora poucos do mundo do software questionem o benefício de um processo de software ágil, organizado ou práticas de engenharia de software sólidas, muitos profissionais e gerentes não sabem o bastante sobre qualquer um dos assuntos.³ Consequentemente, percepções incorretas de processos e práticas levam a decisões inadequadas quando é introduzida uma estrutura

³ Se você está lendo este livro, não será um deles!

SPI. Conclui-se que o elemento-chave de qualquer estratégia SPI é a educação e o treinamento para os profissionais, gerentes técnicos e gerentes seniores que têm contato direto com a organização de software. Deverão ser promovidos três tipos de educação e treinamento:



Tente providenciar treinamento “just-in-time” dirigido para as reais necessidades da equipe de software.

Conceitos e métodos genéricos. Direcionada para gerentes e profissionais, essa categoria explora tanto o processo quanto a prática. A finalidade é fornecer aos profissionais as ferramentas intelectuais de que necessitam para aplicar o processo de software efetivamente e para tomar decisões racionais sobre as melhorias do processo.

Tecnologia e ferramentas específicas. Direcionada primariamente para os profissionais, essa categoria explora tecnologias e ferramentas adotadas para uso local. Por exemplo, se foi escolhida a UML para análise e modelagem de projeto, deve-se providenciar um currículo de treinamento para engenharia de software usando UML.

Comunicação comercial e tópicos relacionados à qualidade. Direcionada para todos os interessados, essa categoria concentra-se nos tópicos leves que ajudam a obter melhor comunicação entre os envolvidos e adotar um foco para melhor qualidade.

Em um contexto moderno, a educação e o treinamento podem ser fornecidos em uma variedade de maneiras diferentes. Tudo, desde podcasts a treinamento baseado na Internet (por exemplo, [QAI08]), até DVDs e cursos em salas de aulas, pode ser oferecido como parte de uma estratégia SPI.

30.2.3 Seleção e justificação



Ao fazer suas escolhas, não deixe de considerar a cultura da sua organização e o nível de aceitação que cada escolha poderá atingir.

Uma vez terminada a atividade de avaliação⁴ e iniciada a educação, uma organização de software deve começar a fazer suas escolhas, que ocorrem durante a *atividade de seleção e justificação* na qual se optam por características de processo e métodos e ferramentas específicos de engenharia de software para preencher o processo de software.

Primeiro você deverá escolher o modelo de processo (Capítulos 2 e 3) que melhor se adapte à sua organização, seus interessados e o software que você produz. Você deverá decidir qual o conjunto de atividades de estrutura a ser aplicado, os principais artefatos que serão produzidos e os pontos de verificação de garantia de qualidade que permitirão à sua equipe acompanhar o progresso. Se a atividade de avaliação de SPI indica uma deficiência específica (por exemplo, não há funções formais de SQA), você deverá concentrar-se nas características do processo que tratam diretamente dessas deficiências.

Em seguida, faça uma divisão do trabalho para cada atividade estrutural (por exemplo, modelagem), definindo o conjunto de tarefas que seriam aplicadas a um projeto típico. Considere também os métodos de engenharia de software que podem ser aplicados para executar essas tarefas. Conforme as escolhas forem feitas, a educação e o treinamento deverão ser coordenados para que o entendimento seja reforçado.

Idealmente, todos trabalham juntos para selecionar vários elementos de processo de tecnologia e poder direcionar-se para a atividade de instalação ou migração (Seção 30.2.4). Na realidade, a seleção pode ser um caminho complicado. Muitas vezes é difícil obter consenso entre os diferentes grupos. Se os critérios para seleção forem estabelecidos por meio de discussões, participantes podem argumentar indefinidamente sobre sua adequação e se determinada escolha realmente atende aos critérios estabelecidos.

Contudo uma má escolha pode prejudicar mais do que ajudar, mas “paralisia por análise” indica pouco progresso (se houver algum) e que os problemas de processo permanecem. Contanto que a característica de processo ou elemento de tecnologia tenha uma boa chance de atender às necessidades de uma organização, muitas vezes é melhor ser objetivo e fazer uma escolha do que esperar por uma solução ótima.

⁴ Na atualidade, a avaliação é uma atividade contínua. Ela é conduzida periodicamente para determinar se a estratégia SPI atingiu seus objetivos imediatos e preparar o cenário para melhorias futuras.

Uma vez feita a escolha, é preciso despendar tempo e dinheiro para fazê-la acontecer na organização, e essas despesas de recursos deverão ser justificadas. Na Seção 30.7 há uma discussão sobre a justificação do custo e o retorno sobre investimento em SPI.

30.2.4 Instalação/migração

Instalação é o primeiro ponto no qual uma organização de software sente os efeitos das mudanças implementadas em consequência do roteiro de SPI. Em alguns casos, é recomendado um processo inteiramente novo para a empresa. Atividades estruturais, ações de engenharia de software e tarefas de trabalho individual devem ser definidas e instaladas como parte de uma nova cultura de engenharia de software. Essas mudanças representam transição organizacional e tecnológica importante e devem ser administradas com muito cuidado.

Em outros casos, mudanças associadas com SPI são relativamente menos importantes, representando modificações pequenas, mas significativas a um modelo de processo existente. Essas mudanças muitas vezes são chamadas de *migração de processo*. Atualmente, muitas organizações de software têm um “processo” em andamento. O problema é que ele não funciona de maneira eficaz. Portanto, a *migração* incremental de um processo (que não funciona tão bem quanto se desejaria) para outro é uma estratégia mais eficaz.

A instalação e migração são na realidade atividades de *redesenho de processo de software* (*software process redesign* - SPR). Scacchi [Sca00] afirma que o “SPR está ligado à identificação, aplicação e refinamento de novas maneiras de melhorar radicalmente e transformar o processo de software”. Quando é iniciada uma abordagem formal ao SPR, consideram-se três diferentes modelos de processo: (1) o processo existente (“da forma como está”), (2) um processo de transição (“daqui para lá”), e (3) o processo-alvo (“o novo”). Se o processo-alvo for significativamente diferente do existente, a única abordagem racional para a instalação é uma estratégia incremental na qual o processo de transição é implementado em etapas. O processo de transição proporciona uma série de pontos intermediários que permitem à cultura da organização de software se adaptar a pequenas alterações durante um período.

30.2.5 Mensuração

Embora ela esteja listada como a última atividade no roteiro de SPI, a *mensuração* ocorre durante toda a SPI. A atividade de mensuração mede o grau segundo o qual as alterações foram criadas e adotadas, o grau segundo o qual essas alterações resultam em software de melhor qualidade ou outros benefícios de processo perceptíveis, e o estado geral do processo e a cultura da organização conforme a SPI progride.

Durante a atividade de mensuração são considerados fatores qualitativos e métricas quantitativas. Do ponto de vista qualitativo, as atitudes da gerência e dos profissionais no passado sobre o processo de software podem ser comparadas com as escolhidas após instalação das mudanças de processo. Métricas quantitativas (Capítulo 25) são coletadas de projetos que usaram o processo de transição ou “o novo” e comparadas com métricas similares coletadas para projetos executados de acordo com o processo “da forma como está”.

30.2.6 Gerenciamento de risco para SPI

A SPI é uma atividade de risco. Na verdade, mais da metade de todos os empreendimentos de SPI terminam em fracasso. As razões do fracasso variam muito e são específicas da organização. Entre os riscos mais comuns estão: falta de suporte gerencial, resistência cultural por parte do pessoal técnico, uma estratégia de SPI mal planejada, uma abordagem excessivamente formal à SPI, escolha de um processo inadequado, falta de interesse por parte dos principais interessados, orçamento inadequado, falta de treinamento do pessoal, instabilidade organizacional e uma infinidade de outros fatores. O papel dos responsáveis pela SPI é analisar os riscos prováveis e desenvolver uma estratégia interna para controlá-los.

PONTO-CHAVE

A SPI muitas vezes falha porque os riscos não foram considerados adequadamente e não houve planejamento de contingência.

Uma organização de software deve administrar o risco em três pontos-chave no processo de SPI [Sta97b]: antes de iniciar o roteiro da SPI, durante a execução das atividades de SPI (avaliação, educação, seleção, instalação) e durante a atividade de avaliação que segue à ocorrência de alguma característica de processo. Em geral, as seguintes categorias podem ser identificadas [Sta97b] para os fatores de risco da SPI: orçamento e custos, conteúdo e resultados práticos, cultura, manutenção de resultados práticos de SPI, missão e metas, gerenciamento organizacional, estabilidade organizacional, interessados no processo, cronograma para o desenvolvimento da SPI, ambiente de desenvolvimento da SPI, processo de desenvolvimento da SPI, gerenciamento de projeto da SPI e pessoal para a SPI.

Em cada categoria, há muitos fatores de risco genéricos. Por exemplo, a cultura organizacional tem uma forte influência sobre o risco. Podem ser identificados os seguintes fatores de risco genéricos⁵ para a categoria cultura [Sta97b]:

- Atitude em relação à mudança, com base em esforços anteriores para mudar
- Experiência com programas de qualidade, nível de sucesso
- Orientação de ações para resolver os problemas *versus* debates políticos
- Uso de fatos para administrar a organização e os negócios
- Paciência com as mudanças, habilidade em empregar o tempo para a socialização
- Orientação para as ferramentas – a expectativa de que as ferramentas possam resolver os problemas
- Nível de “totalidade de planejamento” – habilidade da organização em planejar
- Habilidade dos membros da organização em participar com vários níveis da organização abertamente nas reuniões
- Habilidade dos membros da organização em administrar as reuniões eficientemente
- Nível de experiência em organização com processos definidos

Usando como guia fatores de risco e atributos genéricos, a exposição ao risco é calculada da seguinte maneira:

$$\text{Exposição} = (\text{probabilidade do risco}) \times (\text{perda estimada})$$

Pode ser elaborada uma tabela de riscos (Capítulo 28) para isolar aqueles que requerem mais atenção da gerência.

30.2.7 Fatores críticos de sucesso

Na Seção 30.2.6, destacou-se que a SPI é um empenho arriscado e que a taxa de falha das empresas que tentam melhorar seus processos é assustadoramente alta. Riscos organizacionais, riscos pessoais e riscos de gerenciamento de projeto apresentam desafios para aqueles que empreendem qualquer esforço em SPI. Embora o gerenciamento de risco seja importante, é igualmente importante reconhecer os fatores críticos que levam ao sucesso.

Após examinar 56 organizações de software e seus esforços em SPI, Stelzer e Mellis [Ste99] identificam uma série de fatores críticos de sucesso (*critical success factors* – CSFs) que precisam estar presentes para que a SPI tenha êxito. Nesta seção apresentaremos os cinco principais CSFs.

 **Que fatores críticos de sucesso são cruciais para uma SPI bem-sucedida?**

Comprometimento e suporte da gerência. Assim como a maioria das atividades que causam mudanças organizacionais e culturais, a SPI terá êxito apenas se a gerência estiver envolvida ativamente. Os membros da alta direção deverão reconhecer a importância do software para a empresa e ser patrocinadores ativos do esforço para a SPI. Os gerentes técnicos deverão estar profundamente envolvidos no desenvolvimento da estratégia SPI local. Conforme observam os autores do estudo: “A melhoria do processo de software não é viável

5 Fatores de risco para cada uma das categorias de risco descritas nessa seção podem ser encontrados em [Sta97b].

sem investir tempo, dinheiro e esforços” [Ste99]. O comprometimento e apoio da gerência são essenciais para sustentar esse investimento.

Envolvimento do pessoal. A SPI não pode ser imposta de cima para baixo nem pode ser imposta de fora. Quando se deseja que os esforços em SPI sejam bem-sucedidos, a melhoria tem de ser orgânica — patrocinada por gerentes técnicos e tecnólogos seniores e adotada por profissionais locais.

Integração e entendimento do processo. O processo de software não existe em um vácuo organizacional. Deve ser caracterizado de maneira que esteja integrado a outros processos e requisitos comerciais. Para tanto, os responsáveis pelo trabalho de SPI devem ter um conhecimento e entendimento dos outros processos comerciais. Além disso, devem entender o processo de software “da maneira como está” e apreciar quanta mudança de rumo for tolerável na cultura local.

Uma estratégia SPI personalizada. Não há uma estratégia de SPI genérica. Conforme mencionado neste capítulo, o roteiro da SPI deve ser adaptado ao ambiente local — devem ser considerados a cultura da equipe, variedades de produtos, pontos fortes e fracos locais.

Sólido gerenciamento do projeto de SPI. SPI é um projeto como qualquer outro. Envolve coordenação, cronograma, tarefas paralelas, resultados práticos, adaptação (quando os riscos se tornam realidade), políticas, controle de orçamento e muito mais. Sem um gerenciamento ativo e eficaz, um projeto de SPI está fadado ao fracasso.

30.3 A CMMI

WebRef

Informações completas sobre a CMMI podem ser obtidas no site www.sei.cmu.edu/cmmi/.

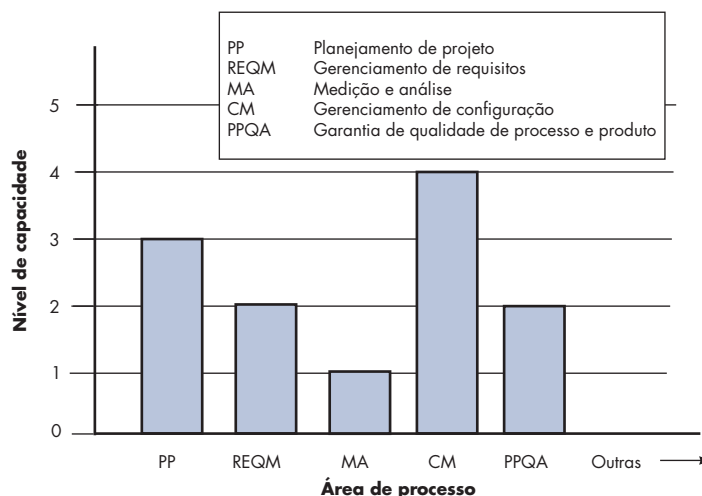
A CMM original foi desenvolvida e atualizada pelo Software Engineering Institute na década de 1990 como uma estrutura SPI completa. Hoje, evoluiu tornando-se a *Capability Maturity Model Integration* (CMMI) [CMM07], um metamodelo de processo abrangente qualificado em uma série de capacidades de sistema e engenharia de software que devem estar presentes à medida que as organizações alcançam diferentes níveis de capacidade e maturidade de processo.

A CMMI representa um metamodelo de processo de duas maneiras diferentes: (1) como um modelo “contínuo” e (2) como um modelo “encenado”. O metamodelo CMMI contínuo descreve um processo em duas dimensões conforme está ilustrado na Figura 30.2. Cada área de processo (por exemplo, planejamento de projeto ou gerenciamento de requisitos) é formalmente avaliada

FIGURA 30.2

Perfil de capacidade de área de processo CMMI

Fonte: (Phi02)



em relação a metas e práticas específicas e classificada de acordo com os seguintes níveis de capacidade:

Nível 0: Incompleto – a área de processo (por exemplo, gerenciamento de requisitos) não funciona ou não atinge todas as metas e objetivos definidos pela CMMI para a capacidade nível 1 para a área de processo.

Nível 1: Executado – todas as metas específicas da área de processo (definida pela CMMI) foram satisfeitas. Estão sendo executadas as tarefas necessárias para produzir os artefatos definidos.

Nível 2: Controlada – todos os critérios do nível de capacidade 1 foram satisfeitos. Além disso, todo o trabalho associado com a área de processo está de acordo com uma política definida em termos de organização; todas as pessoas que estão fazendo o trabalho têm acesso a recursos adequados para executar o trabalho; os interessados são envolvidos ativamente na área de processo conforme necessário; todas as tarefas e produtos são “monitorados, controlados, e revisados; e são avaliados quanto à conformidade com a descrição de processo” [CMM07].

Nível 3: Definido – todos os critérios do nível de capacidade 2 foram satisfeitos. Além disso, o processo é “adaptado com base no conjunto de processos padronizados da organização de acordo com as regras de adaptação da organização, e dos produtos acabados, medidas e outras informações de melhoria de processo para agregar valores ao processo organizacional” [CMM07].

Nível 4: Controlado quantitativamente – todos os critérios do nível de capacidade 3 foram satisfeitos. Além disso, a área de processo é controlada e melhorada usando medição e avaliação quantitativa. “São estabelecidos objetivos quantitativos para qualidade e desempenho de processo e utilizados como critérios no controle do processo” [CMM07].

Nível 5: Otimizado – todos os critérios do nível de capacidade 4 foram satisfeitos. Além disso, a área de processo é adaptada e otimizada usando meios quantitativos (estatísticos) para atender à mudança de necessidades do cliente e melhorar continuamente a eficiência da área de processo em consideração.

A CMMI define cada área de processo em termos de “metas específicas” e as “práticas específicas” necessárias para atingir essas metas. As *metas específicas* estabelecem as características que devem existir para que as atividades envolvidas por uma área de processo sejam eficazes. *Práticas específicas* refinam uma meta transformando-a em uma série de atividades relacionadas ao processo.

Por exemplo, **planejamento de projeto** é uma das oito áreas de processo definidas pela CMMI para a categoria “gerenciamento de projeto”.⁶ As metas específicas (*specific goals* – SG) e as práticas específicas associadas (*specific practices* – SP), definidas para o **planejamento de projeto** são [CMM07]:

SG 1 Estabelecer Estimativa

- SP 1.1-1 Estimar o Escopo do Projeto
- SP 1.2-1 Estabelecer Estimativas de Atributos de Produto e Tarefa
- SP 1.3-1 Definir o Ciclo de Vida do Projeto
- SP 1.4-1 Determinar Estimativas de Trabalho e Custo

SG 2 Desenvolver um Plano de Projeto

- SP 2.1-1 Estabelecer o Orçamento e o Cronograma
- SP 2.2-1 Identificar os Riscos do Projeto



Toda organização deve lutar para atingir o objetivo da CMMI. No entanto, a implementação de todo o aspecto do modelo pode ser desastrosa na sua situação.

WebRef

Informações completas, bem como download de uma versão da CMMI, podem ser obtidas no site www.sei.cmu.edu/cmmi/.

⁶ Outras áreas de processo para “gerenciamento de projeto” incluem: monitoramento e controle de projeto, gerenciamento de acordos com fornecedores, gerenciamento integrado de projeto para IPPD, gerenciamento de risco, equipe integrada, gerenciamento integrado de fornecedor e gerenciamento quantitativo de projeto.

- SP 2.3-1 Planejar o Gerenciamento de Dados
- SP 2.4-1 Elaborar Plano para Recursos de Projeto
- SP 2.5-1 Elaborar Plano para Conhecimento e Habilidades Necessárias
- SP 2.6-1 Elaborar Plano para Envolvimento dos Interessados
- SP 2.7-1 Estabelecer o Plano de Projeto

SG 3 Obter Comprometimento com o Plano

- SP 3.1-1 Rever Planos Que Afetam o Projeto
- SP 3.2-1 Reconciliar Trabalho e Níveis de Recursos
- SP 3.3-1 Obter Comprometimento com o Plano

Além das metas e práticas específicas, a CMMI também define um conjunto de cinco metas genéricas e práticas relacionadas a cada área de processo. Cada uma delas corresponde a um dos cinco níveis de capacidade. Portanto, para atingir determinado nível de capacidade, a meta genérica para aquele nível e as práticas genéricas que correspondem àquela meta devem ser atingidas. As metas genéricas (*generic goals* – GG) e práticas genéricas (*generic practices* – GP) para a área de processo de **planejamento de projeto** são [CMM07]:

GG 1 Atingir Metas Específicas

- GP 1.1 Executar Práticas Básicas

GG 2 Instituir um Processo Controlado

- GP 2.1 Estabelecer uma Política Organizacional
- GP 2.2 Planejar o Processo
- GP 2.3 Fornecer os Recursos
- GP 2.4 Atribuir Responsabilidades
- GP 2.5 Treinar o Pessoal
- GP 2.6 Gerenciar Configurações
- GP 2.7 Identificar e Envolver Interessados Importantes
- GP 2.8 Monitorar e Controlar o Processo
- GP 2.9 Avaliar Objetivamente a Adesão
- GP 2.10 Rever o Status com a Alta Gerência

GG 3 Instituir um Processo Definido

- GP 3.1 Estabelecer um Processo Definido
- GP 3.2 Coletar Informações de Melhorias

GG 4 Instituir um Processo Controlado Quantitativamente

- GP 4.1 Estabelecer Objetivos Quantitativos do Processo
- GP 4.2 Estabilizar Desempenho de Subprocesso

GG 5 Instituir um Processo de Otimização

- GP 5.1 Garantir a Melhora Contínua do Processo
- GP 5.2 Corrigir as Causas Importantes dos Problemas

O modelo CMMI apresentado define as mesmas áreas de processo, metas e práticas do modelo contínuo. A diferença principal é que o modelo apresentado define cinco níveis de maturidade, em vez de cinco níveis de capacidade. Para atingir um nível de maturidade, as metas específicas e as práticas associadas com uma série de áreas de processo devem ser atingidas. A relação entre níveis de maturidade e áreas de processo é apresentada na Figura 30.3.

INFORMAÇÕES

**A CMMI – Fazemos ou não fazemos?**

A CMMI é um metamodelo de processo. Ela define (em mais de 700 páginas) as características de processo que devem existir se uma organização de software quiser estabelecer um processo de software completo. A questão debatida por mais de uma década é: “A CMMI é opressiva?”. Como muitas coisas na vida (e em software), a resposta não é um simples “sim” ou “não”.

O espírito da CMMI deverá ser sempre adotado. Com o risco da simplificação, argumenta que o desenvolvimento de software deve ser encarado com seriedade – deve ser planejado amplamente, precisa ser controlado com uniformidade, deve ser acompanhado com precisão e conduzido profissionalmente. Deve concentrar-se nas necessidades dos patrocinadores/interessados pelo projeto, nas habilidades dos engenheiros de software e na qualidade do produto final. Essas ideias são indiscutíveis.

Os requisitos detalhados da CMMI deverão ser considerados seriamente se uma organização cria sistemas grandes e

complexos que envolvem dezenas ou centenas de pessoas por muitos meses ou anos. Pode ocorrer que a CMMI seja “na medida certa” em tais situações, se a cultura organizacional for favorável a modelos de processo-padrão e a gerência estiver comprometida em torná-la um sucesso. No entanto, em outras situações, a CMMI pode ser demais para uma organização poder assimilá-la. Isso significa que a CMMI é “ruim” ou “demasiadamente burocrática” ou “à moda antiga?”. Não ... não é. Significa que o que é certo para uma cultura organizacional pode não ser certo para outra.

A CMMI é uma conquista significativa na engenharia de software. Proporciona uma discussão abrangente das atividades e ações que devem existir quando uma organização cria software de computador. Mesmo que a empresa prefira não adotar seus detalhes, toda equipe de software deveria aderir a esse espírito e aprender com a discussão de processo e prática de engenharia de software.

FIGURA 30.3

Áreas de processo necessárias para atingir um nível de maturidade

Fonte: (Phi02)

Nível	Foco	Áreas de Processo
Otimizante	<i>Melhoria contínua do processo</i>	Inovação organizacional e distribuição (deployment) Análise causal e resolução
Controlado quantitativamente	<i>Gerenciamento quantitativo</i>	Desempenho de processo organizacional Gerenciamento quantitativo de projeto
Definido	<i>Padronização de processo</i>	Desenvolvimento de requisitos Solução técnica Integração de produto Verificação Validação Foco no processo organizacional Definição de processo organizacional Treinamento organizacional Gerenciamento de projeto integrado Gerenciamento de fornecimento integrado Gerenciamento de risco Análise de decisão e resolução Ambiente organizacional para integração Equipe integrada
Repetível	<i>Gerenciamento básico de projeto</i>	Gerenciamento de requisitos Planejamento de projeto Monitoração e controle de projeto Gerenciamento de acordo com fornecedor Medição e análise Garantia de qualidade de processo e produto Gerenciamento de configuração
Executado		

30.4 A CMM DAS PESSOAS

PONTO-CHAVE

A CMM das Pessoas sugere práticas que melhoram a competência e cultura da força de trabalho.

Um processo de software, não importa quão bem seja concebido, não terá sucesso sem profissionais talentosos e motivados. O *Modelo de Maturidade da Capacidade das Pessoas* (*People Capability Maturity Model* – CMM) “é um roteiro para implementar práticas de trabalho que aperfeiçoam continuamente a capacidade dos profissionais de uma organização” [Cur02]. Desenvolvida no decorrer da década de 1990 e refinada nos anos seguintes, o objetivo da CMM das Pessoas (People CMM) é estimular a melhoria contínua do conhecimento genérico da força de trabalho (chamado de “competências núcleo”), habilidades específicas de engenharia de software e gerenciamento de projeto (chamado de “competências da força de trabalho”) e habilidades relacionadas ao processo.

Assim como a CMM, a CMMI e estruturas SPI relacionadas, a CMM das Pessoas define um conjunto de cinco níveis de maturidade organizacional que proporcionam uma indicação da sofisticação relativa das práticas e processos da força de trabalho. Esses níveis de maturidade [CMM08] estão ligados à existência (dentro de uma organização) de um conjunto de áreas-chave de processo (*key process areas* - KPAs). Uma visão geral dos níveis organizacionais e KPAs relacionadas está na Figura 30.4.

A CMM das Pessoas complementa qualquer estrutura SPI estimulando uma organização a cultivar e melhorar seu bem mais importante – as pessoas. Sendo importante, estabelece uma atmosfera na força de trabalho que permite a organização de software “atrair, desenvolver e preservar talentos notáveis” [CMM08].

FIGURA 30.4

Áreas de processo para CMM das Pessoas

Nível	Foco	Áreas de Processo
Otimizante	<i>Melhoria contínua</i>	Inovação contínua da força de trabalho Alinhamento do desempenho organizacional Melhora contínua da capacidade
Gerenciado	<i>Identifica e desenvolve conhecimento, prática e habilidades</i>	Tutela Gerenciamento da capacidade organizacional Gerenciamento do desempenho quantitativo Propriedades baseadas na competência Grupos de trabalho fortalecidos Integração de competência
Definido	<i>Quantifica e administra conhecimento, prática e habilidades</i>	Cultura participativa Desenvolvimento de grupo de trabalho Práticas baseadas na competência Desenvolvimento de carreira Desenvolvimento de competência Planejamento da força de trabalho Análise de competência
Repetível	<i>Práticas básicas de gerenciamento de pessoas que podem ser repetidas</i>	Compensação Treinamento e desenvolvimento Gerenciamento de desempenho Ambiente de trabalho Comunicação e coordenação de pessoal
Inicial	<i>Práticas inconsistentes</i>	

30.5 OUTRAS ESTRUTURAS SPI

Embora a CMM e CMMI do Software Engineering Institute (SEI) sejam as estruturas SPI mais amplamente aplicadas, têm sido propostas e estão em uso muitas alternativas⁷. Entre elas, destacam-se:

- **SPICE** – iniciativa internacional para suportar a avaliação de processo da ISO e padrões de processo de ciclo de vida [SPI99]
- **ISO/IEC 15504** para Avaliação de Processo (Software) [ISO08]
- **Bootstrap** – uma estrutura SPI para organizações de pequeno e médio porte em conformidade com SPICE [Boo06]
- **PSP e TSP** – estruturas SPI individuais e específicas de equipe ([Hum97], [Hum00]) que se concentram no processo em detalhes, uma abordagem mais rigorosa do desenvolvimento de software combinada com a medição
- **TickIT** – um método de auditoria [Tic05] que avalie se uma organização está em conformidade com a Norma ISO 9001:2000

Nos próximos parágrafos há uma descrição resumida das estruturas SPI. Se houver interesse, para cada uma delas há uma grande variedade de material impresso e baseado na Web.

 Além da CMM, há outras estruturas SPI que podem ser consideradas?

SPICE. O modelo SPICE (*Software Process Improvement and Capability dEtermination - Melhoria de Processo de Software e Determinação de Capacidade*) proporciona uma estrutura de avaliação SPI compatível com a ISO 15504:2003 e ISO 12207. O conjunto de documentos SPICE [SDS08] apresenta uma estrutura SPI completa, incluindo um modelo para gerenciamento de processo, diretrizes para conduzir uma avaliação e classificação do processo em consideração, construção, seleção e uso dos instrumentos e ferramentas de avaliação e treinamento para os avaliadores.

Bootstrap. A estrutura SPI *Bootstrap* “foi desenvolvida para assegurar a conformidade com a norma ISO emergente para avaliação e melhoria de processo de software (SPICE) e para alinhar a metodologia com a ISO 12207” [Boo06]. O objetivo da Bootstrap é examinar um processo de software usando um conjunto de melhores práticas de engenharia de software como base para a avaliação. Assim como a CMMI, a Bootstrap proporciona um nível de maturidade de processo por meio dos resultados de questionários que reúnem informações sobre o processo de software e projetos de software “da forma como está”. As diretrizes SPI baseiam-se no nível de maturidade e nas metas organizacionais.

PSP e TSP. Embora a SPI seja caracterizada em geral como uma atividade organizacional, não há razão para que uma melhoria de processo não possa ser conduzida em nível individual ou de equipe. Tanto a PSP quanto a TSP (Capítulo 2) enfatizam a necessidade de continuamente coletar dados sobre o trabalho que está em execução e usar esses dados para desenvolver estratégias para melhorias. Watts Humphrey [Hum97], o desenvolvedor de ambos os métodos, comenta:

A PSP [e TSP] lhe mostrarão como planejar e acompanhar o seu trabalho e como produzir consistentemente software de alta qualidade. Usando PSP [e TSP] você obterá os dados que indicam a eficácia do seu trabalho e que identificam seus pontos fortes e fracos... Para uma carreira bem-sucedida e bem remunerada, você precisa conhecer suas experiências e habilidades, lutar para melhorá-las e concentrar seu talento exclusivamente no trabalho.

TickIT. O método de auditoria Ticket assegura a conformidade com a norma ISO 9001:2000 para Software – uma norma genérica que se aplica a qualquer organização que queira melhorar

“As organizações de software têm mostrado deficiências significativas em tirar proveito das experiências adquiridas com projetos executados.”

NASA

⁷ É razoável argumentar que algumas dessas estruturas não são muito “alternativas” já que são abordagens complementares à SPI. Uma tabela abrangente de muitas outras estruturas SPI pode ser encontrada no site www.geocities.com/lbu_measure/spi/spi.htm#p2.

a qualidade geral dos produtos, sistemas ou serviços que ela fornece. Portanto, a norma é diretamente aplicável a organizações e empresas de software.

A estratégia subjacente sugerida pela ISO 9001:2000 é descrita da seguinte maneira [ISO01]:

ISO 9001:2000 destaca a importância para uma organização em identificar, implementar, administrar e melhorar continuamente a eficácia dos processos necessários para o sistema de gerenciamento de qualidade, e para administrar as interações desses processos com o fim de atingir os objetivos da organização... A eficácia e eficiência do processo podem ser avaliadas por meio de processos de revisão interna e externa e podem ser avaliadas em uma escala de maturidade.

WebRef

Um excelente resumo da ISO 9001: 2000 pode ser encontrado no site <http://praxiom.com/iso-9001.htm>.

A norma ISO 9001:2000 adotou um ciclo “planejar-fazer-verificar-agir” aplicado aos elementos de gerenciamento de qualidade de um projeto de software. Em um contexto de software, o “planejar” estabelece os objetivos do processo, atividades e tarefas necessárias para obter software de alta qualidade e como resultado a satisfação do cliente. “Fazer” implementa o processo de software (incluindo tanto atividades de estrutura quanto de apoio). “Verificar” monitora e mede o processo para garantir que todos os requisitos estabelecidos para gerenciamento da qualidade tenham sido atingidos. “Agir” inicia as atividades de melhoramento do processo de software que contribuem continuamente para melhorar o processo. TickIT pode ser usado em todo o ciclo “planejar-fazer-verificar-agir” para assegurar que o progresso SPI esteja ocorrendo. Os auditores TickIT avaliam a aplicação do ciclo como um precursor da certificação ISO 9001:2000. Para uma discussão detalhada da ISO 9001:2000 e TickIT, consulte [Ant06], [Tri05], ou [Sch03].

30.6 RETORNO SOBRE INVESTIMENTO EM SPI

A SPI é um trabalho pesado e requer investimento substancial em dinheiro e profissionais. Os administradores que aprovam o orçamento e recursos destinados à SPI invariavelmente farão a seguinte pergunta: “Como posso saber se obtivemos um retorno razoável do dinheiro que gastamos?”.

Em nível qualitativo, os proponentes da SPI argumentam que um processo de software melhorado levará a uma qualidade superior. Eles sustentam que um processo melhorado resultará na implementação de filtros de qualidade superior (resultando em menor propagação de defeitos), melhor controle das alterações (resultando em menor caos no projeto) e menos retrabalho técnico (resultando em menor custo e melhor prazo de entrega para o mercado). Mas podem esses benefícios qualitativos ser traduzidos em resultados quantitativos? A equação clássica do retorno sobre o investimento (*return on investment* – ROI) é:

$$ROI = \left[\frac{\Sigma(\text{benefícios}) - \Sigma(\text{custos})}{\Sigma(\text{custos})} \right] \times 100\%$$

em que

benefícios incluem as economias em custos associados a melhor qualidade do produto (menos defeitos), menos retrabalho, redução do trabalho associado com alterações e lucro proveniente de uma entrega mais rápida ao mercado.

custos incluem custos SPI diretos (por exemplo, treinamento, medição) e custos indiretos associados a maior ênfase no controle de qualidade e atividades de gerenciamento de mudanças e uma aplicação mais rigorosa dos métodos de engenharia de software (por exemplo, a criação de um modelo de projeto).

Na prática, esses benefícios quantitativos e custos são muitas vezes difíceis de medir com precisão, e todos dependem de uma interpretação. Mas isso não significa que uma organização de software deverá executar um programa SPI sem cuidadosa análise dos custos e benefícios

que se acumulam. Um tratamento abrangente do retorno de investimento para a SPI pode ser encontrado em um livro especial de David Rico [Ric04].

30.7 TENDÊNCIAS DA SPI

Durante as últimas duas décadas, muitas empresas tentaram melhorar suas práticas de engenharia de software aplicando uma estrutura SPI para promover mudança organizacional e transição de tecnologia. Conforme observamos antes neste capítulo, mais da metade fracassou nessa tentativa. Independentemente do sucesso ou falha, tudo custa muito. David Rico [Ric04] relata que uma aplicação típica de uma estrutura SPI como a SEI CMM pode custar entre \$ 25 mil e \$ 70 mil por pessoa e levar anos para se completar! Não é surpresa o fato de que o futuro da SPI deve dar ênfase a uma abordagem menos custosa e menos demorada.

Para ser eficaz no mundo do desenvolvimento de software do século 21, as futuras estruturas SPI devem se tornar significativamente mais ágeis. Em vez de um foco organizacional (que pode levar anos para se completar com sucesso), os esforços de SPI contemporâneos devem concentrar-se no nível de projeto, trabalhando para melhorar um processo de equipe em semanas, não em meses ou anos. Para obter resultados significativos (mesmo em nível de projeto) em curto prazo, modelos complexos de estrutura podem ser substituídos por modelos mais simples. Em lugar de dezenas de práticas-chave e centenas de práticas suplementares, uma estrutura SPI ágil deve dar ênfase apenas a poucas práticas críticas (por exemplo, análogas às atividades de estrutura discutidas neste livro).

Qualquer tentativa em SPI demanda uma força de trabalho com conhecimento, mas as despesas com educação e treinamento podem ser altas e deverão ser minimizadas (e enxugadas). Em vez de cursos em salas de aula, (caros e demorados), trabalhos de SPI futuros deverão utilizar o treinamento baseado na Web, destinado a práticas críticas. Ao contrário de tentativas avançadas para mudar a cultura organizacional (com todos os perigos políticos que surgem), a mudança cultural deverá ocorrer com um pequeno grupo de cada vez até alcançar um estágio importante.

O trabalho de SPI das duas últimas décadas tem um mérito significativo. As estruturas e modelos desenvolvidos representam valores intelectuais importantes para a comunidade de engenharia de software. Mas esses valores direcionam futuras tentativas em SPI não se tornando dogma recorrente, mas servindo como base para modelos SPI melhores, mais simples e mais ágeis.

30.8 RESUMO

Uma estrutura para melhorar o processo de software define as características que devem estar presentes quando se quer atingir um processo de software eficaz, um método de avaliação que ajude a determinar se aquelas características estão presentes e uma estratégia para ajudar a organização de software na implementação das características de processo consideradas fracas ou ausentes. Independentemente do grupo que apoia a SPI, o objetivo é melhorar a qualidade do processo e, como consequência, melhorar a qualidade do software e os prazos.

O modelo de maturidade de processo proporciona uma indicação geral da “maturidade de processo” exibida por uma organização de software. Proporciona uma sensação qualitativa sobre a eficiência relativa do processo de software que está em uso no momento.

O roteiro da SPI começa com a avaliação – uma série de atividades que revelam os pontos fracos e fortes como sua organização aplica o processo de software e as práticas de engenharia de software que fazem parte do processo. Desse modo, a empresa pode desenvolver um plano geral de SPI.

Um dos elementos-chave de qualquer plano SPI é a educação e o treinamento, uma atividade que se concentra na melhora do nível de conhecimento dos gerentes e profissionais. Depois que

o pessoal se torna experiente nas tecnologias de software atuais, começa a seleção e a justificação. Essas tarefas levam a escolhas sobre a arquitetura do processo de software, os métodos que fazem parte dela e as ferramentas que a suportam. Instalação e avaliação são atividades de SPI que causam mudanças no processo e examinam sua eficácia e impacto.

Para melhorar o processo de software, a organização deve possuir as seguintes características: comprometimento e suporte da gerência para com a SPI, envolvimento do pessoal durante todo o processo de SPI, integração do processo na cultura organizacional geral, uma estratégia SPI que tenha sido personalizada para as necessidades locais e uma administração sólida do projeto de SPI.

Hoje há uma série de estruturas SPI em uso: CMM e CMMI da SEI são amplamente utilizadas. A People CMM foi personalizada para avaliar a qualidade da cultura organizacional e os profissionais envolvidos. SPICE, Bootstrap, PSP, TSP e TickIT são estruturas adicionais que podem levar a uma SPI eficaz.

SPI é trabalho sério e requer investimento substancial em dinheiro e pessoas. Para garantir a obtenção de um retorno sobre o investimento razoável, a organização deve medir os custos associados com a SPI e os benefícios que podem ser atribuídos.

PROBLEMAS E PONTOS A PONDERAR

- 30.1.** Por que as organizações de software usualmente lutam com dificuldade quando aderem a um esforço para melhorar o processo local de software?
- 30.2.** Descreva o conceito de “maturidade de processo” com suas próprias palavras.
- 30.3.** Faça uma pesquisa (verifique no site da SEI) e determine a distribuição da maturidade de processo para organizações de software nos Estados Unidos e no mundo todo.
- 30.4.** Você trabalha para uma pequena organização de software – apenas 11 pessoas estão envolvidas no desenvolvimento de software. A SPI é adequada a sua empresa? Explique sua resposta.
- 30.5.** Avaliação é semelhante a um exame anual de saúde. Usando essa metáfora, descreva a atividade de avaliação SPI.
- 30.6.** Qual a diferença entre um processo “como está”, um processo “desta para próxima etapa”, e um processo “vir a ser”?
- 30.7.** Como é aplicado o gerenciamento de risco no contexto da SPI?
- 30.8.** Selecione um dos fatores críticos de sucesso na Seção 30.2.7. Faça uma pesquisa e escreva um pequeno artigo sobre como pode ser alcançado.
- 30.9.** Faça uma pesquisa e explique como a CMMI difere de sua antecessora, a CMM.
- 30.10.** Selecione uma das estruturas SPI discutidas na Seção 30.5, e redija um pequeno artigo descrevendo-a em mais detalhes.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Um dos recursos mais facilmente acessíveis e abrangentes de informações sobre SPI foi desenvolvido pelo Software Engineering Institute e está disponível no site www.sei.cmu.edu. O site do SEI contém centenas de artigos, estudos e descrições detalhadas de estruturas SPI.

Durante os últimos anos, muitos bons livros foram acrescentados a uma ampla literatura desenvolvida durante as duas últimas décadas. Land (*Jumpstart CMM/CMMI Software Process Improvements*, Wiley-IEEE Computer Society, 2007) apresenta os requisitos definidos como parte dos padrões de engenharia de software SEI CMM e CMMI com IEEE, com ênfase na interseção de processo e prática. Mutafelija e Stromberg (*Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech House Publishers, 2007) discutem as estruturas ISO 9001:2000 e CMMI SPI e a “sinergia” entre elas. Conradi e seus colegas (*Software Process Improvement: Re-*

sults and Experience from the Field, Springer, 2006) apresentam os resultados de uma série de estudos de caso e experimentos relacionados com SPI. Van Loon (*Process Assessment and Improvement: A Practical Guide for Managers, Quality Professionals and Assessors*, Springer, 2006) discute a SPI no contexto da ISO/IEC 15504. Watts Humphrey (*PSP*, Addison-Wesley, 2005, and *TSP*, Addison-Wesley, 2005) trata de sua estrutura SPI Personal Team Process e Team Software Process em dois livros. Fantina (*Practical Software Process Improvement*, Artech House Publishers, 2004) traz instruções pragmáticas com ênfase em CMMI/CMM.

Uma grande variedade de fontes de informação sobre melhoria de processo de software está disponível na Internet. Uma lista atualizada das referências na Web relevantes a SPI pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

TENDÊNCIAS EMERGENTES NA
ENGENHARIA DE SOFTWARECONCEITOS-
-CHAVE

aberto	706
blocos básicos.	708
ciclo da excelência	703
ciclo de vida da	
inovação.	702
complexidade	705
desenvolvimento	
colaborativo	712
desenvolvimento	
controlado por	
modelo.	712
desenvolvimento	
motivado por teste .	714
engenharia de	
requisitos	713
evolução da	
tecnologia.	702
ferramentas	701
projeto	
pós-moderno.	714
requisitos	
emergentes.	707
rumos da	
tecnologia.	710
software de	
código aberto	706
tendências leves ..	703

Durante a história relativamente recente da engenharia de software, profissionais e pesquisadores desenvolveram uma série de modelos de processo, métodos técnicos e ferramentas automatizadas em um esforço para apoiar a mudança na maneira como criamos software para computador. Apesar de a experiência indicar o contrário, há um desejo implícito de se encontrar a “solução mágica” — o processo mágico ou tecnologia transcendente que nos permitirá criar facilmente sistemas grandes e complexos, sem confusão, enganos e atrasos — sem os vários problemas que continuam a povoar o trabalho de software.

Mas a história indica que nossa busca pela solução mágica parece estar fadada ao fracasso. Novas tecnologias são introduzidas regularmente, apresentadas como sendo a “solução” para muitos dos problemas que os engenheiros de software enfrentam e incorporadas nos projetos grandes e pequenos. Críticos da indústria exageram a importância dessas “novas” tecnologias de software, os conhecedores (*cognoscenti*) da comunidade do software as adotam com entusiasmo, e, por fim, elas desempenham um papel no mundo da engenharia de software, mas tendem a não produzir o resultado esperado, e, como consequência, a busca continua.

Mili e Cowan [Mil00b] comentam sobre os desafios que enfrentamos ao tentarmos isolar tendências significativas na tecnologia:

Que fatores determinam o sucesso de uma tendência? O que caracteriza as tendências bem-sucedidas: Seu mérito técnico? Sua habilidade para abrir novos mercados? Sua habilidade para alterar a economia dos mercados existentes?

Qual é o ciclo de vida de uma tendência? Embora a visão tradicional de que as tendências evoluem ao longo de um ciclo de vida bem definido e previsível, que vai da pesquisa a um produto acabado, por meio de um processo de transferência, descobrimos que muitas encurtaram esse ciclo ou seguiram outro.

Com que antecedência pode uma tendência bem-sucedida ser identificada? Se soubermos identificar fatores de sucesso e/ou se entendermos o ciclo de vida de uma tendência, poderemos detectar sinais precoces de êxito. Retoricamente, buscamos a habilidade para reconhecer a próxima tendência antes dos outros.

PANORAMA

O que é? Ninguém pode prever o futuro com certeza absoluta. Mas é possível avaliar tendências na área de engenharia de software e a partir dessas tendências sugerir direções possíveis para a tecnologia. É isso que eu tento fazer neste capítulo.

Quem realiza? Qualquer um que deseje dedicar seu tempo para se manter atualizado sobre os problemas da engenharia de software pode tentar prever a direção futura da tecnologia.

Por que é importante? Porque os reis na antiguidade consultavam os adivinhos? Porque grandes corporações multinacionais contratam firmas de consultoria e se esforçam para preparar previsões? Porque uma grande parte do público lê horóscopos? Queremos sempre saber o que vai acontecer para nos preparar.

Quais são as etapas envolvidas? Não há uma fórmula para prever o caminho à frente. Tentamos fazer isso coletando dados, organizando esses dados para que nos forneçam informações úteis, examinando associações sutis para extrair conhecimento, e desse conhecimento sugerir prováveis tendências que preveem como as coisas serão em algum tempo futuro.

Qual é o artefato? Uma visão do futuro próximo que pode ser ou não correta.

Como garantir que o trabalho foi realizado corretamente? Prever o caminho à frente é uma arte, não uma ciência. De fato, é muito raro que uma previsão séria sobre o futuro esteja absolutamente certa ou completamente errada (com exceção, felizmente, das previsões do fim do mundo). Nós procuramos tendências e tentamos extrapolá-las. Podemos avaliar a precisão dessa extrapolação somente com o passar do tempo.

? Quais são as “grandes questões” quando consideramos a evolução da tecnologia?

Quais os aspectos controláveis da evolução? Podem as corporações usar sua influência no mercado para impor tendências? Pode o governo usar seus recursos para impor tendências? Qual é o papel das normas e padrões na definição das tendências? Uma cuidadosa análise comparativa entre Ada e Java, por exemplo, poderia ser esclarecedora nesse aspecto?

Não há uma resposta simples a essas questões e não há dúvida de que as tentativas para identificar tecnologias significativas são, no melhor caso, mediocres.

Em edições anteriores deste livro (durante os últimos 30 anos), tenho discutido tecnologias emergentes e seu impacto sobre a engenharia de software. Algumas foram amplamente adotadas, outras nunca alcançaram seu potencial. Minha conclusão: as tecnologias vêm e vão; as tendências reais que devemos explorar são mais flexíveis. Em outras palavras, o progresso na engenharia de software será orientado pelas tendências nos negócios, organizações, mercado e tendências culturais. Estas levam à inovação tecnológica.

Neste capítulo, examinaremos algumas tendências tecnológicas na engenharia de software, mas minha ênfase será sobre algumas tendências nos negócios, nas organizações, no mercado e culturais que podem ter influência importante sobre a tecnologia de engenharia de software durante os próximos 10 ou 20 anos.

31.1 EVOLUÇÃO DA TECNOLOGIA

Em uma obra fascinante que oferece uma visão atraente sobre como as tecnologias de computação (e outras tecnologias relacionadas) irão evoluir, Ray Kurzweil [Kur05] afirma que a evolução tecnológica é similar à biológica, mas ocorre a uma velocidade cuja magnitude é muitas vezes maior. A evolução (seja ela biológica, seja tecnológica) ocorre em resultado de uma realimentação positiva — “os métodos mais capacitados resultantes de um estágio do progresso evolucionário são usados para criar o próximo estágio” [Kur06].

As grandes questões para o século 21 são: (1) Quão rápido uma tecnologia evolui? (2) Quão significativos são os efeitos da realimentação positiva? (3) Quão profundas serão as mudanças resultantes?

“Previsões são muito difíceis de fazer, especialmente quando se trata de futuro.”

Mark Twain

Quando é introduzida uma tecnologia bem-sucedida, o conceito inicial transforma-se em “ciclo de vida de inovação” razoavelmente previsível [Gai95], ilustrado na Figura 31.1. Na fase *avanço*, um problema é identificado e tentativas repetidas são realizadas em busca de uma solução viável. Em algum ponto, aparece uma promessa de solução. O trabalho inicial de avançar é reproduzido na fase *replicador* e ganha um uso mais amplo. O *empirismo* leva à criação de regras que regem o uso da tecnologia, e o sucesso repetido leva a uma *teoria* de uso mais amplo seguida pela criação de ferramentas automatizadas durante a fase da *automação*. Por fim, a tecnologia amadurece e passa a ser amplamente utilizada.

Devemos observar que muitas pesquisas e tendências tecnológicas nunca atingem a maturidade. Na verdade, a grande maioria das tecnologias “promissoras” no domínio da engenharia de software suscita um grande interesse por alguns anos e depois passa a ser usada por um grupo dedicado de usuários. Isso não significa que não tenham mérito, mas que o caminho através da inovação é longo e difícil.

Kurzweil [Kur05] concorda que as tecnologias de computação evoluem através de uma “curva-S”: apresentam um crescimento relativamente lento durante os anos de formação da tecnologia, rápida aceleração durante a fase de crescimento e depois um período de nivelamento quando atinge seus limites. Mas a tecnologia da computação e outras relacionadas têm mostrado um crescimento explosivo (exponencial) durante os estágios centrais (veja a Figura 31.1) e assim continuarão. Além disso, quando uma curva-S termina, outra a substitui com um crescimento ainda mais explosivo durante seu período de crescimento.¹ Hoje, estamos no joelho da curva-S das modernas tecnologias de computação — na transição entre o crescimento inicial e o crescimento explosivo

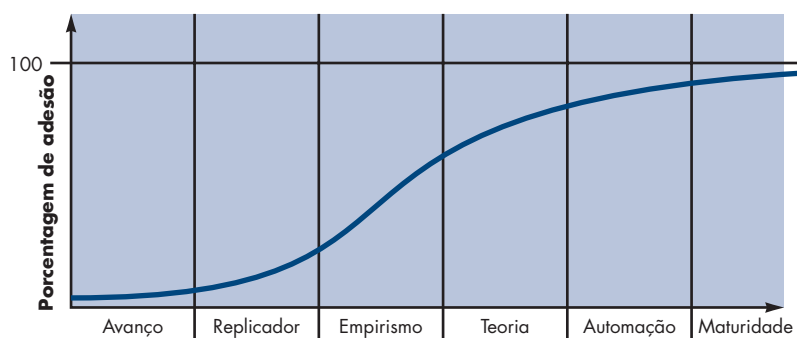
PONTO-CHAVE

A tecnologia da computação está evoluindo a uma taxa exponencial, e seu crescimento pode logo se tornar explosivo.

¹ Por exemplo, os limites dos circuitos integrados podem ser esgotados na próxima década, mas essa tecnologia pode ser substituída pelas tecnologias de computação molecular e por outra curva-S acelerada.

FIGURA 31.1

Um ciclo de vida da evolução tecnológica



que deve se seguir. A implicação é que durante os próximos 20 a 40 anos, veremos mudanças significativas (até mesmo impressionantes) na capacidade de computação. As décadas vindouras resultarão em mudanças de grande magnitude na computação em termos de velocidade, tamanho, capacidade e consumo de energia (para citar apenas algumas características).

Kurzweil [Kur05] sugere que, em 20 anos, a evolução tecnológica irá acelerar a um passo cada vez mais rápido, chegando finalmente à era de inteligência não biológica que se combinará com a inteligência humana e a ampliará de maneira fascinante.

E tudo isso, não importa como evolua, necessitará de software e sistemas que na comparação fazem nossos esforços atuais parecerem infantis. Por volta de 2040, uma combinação de computação extrema, nanotecnologia, redes com larguras de banda extremamente altas e robótica nos levará a um mundo diferente.² O software, possivelmente na forma em que ainda não podemos compreender, continuará existindo no centro desse mundo novo. A engenharia de software não irá acabar.

31.2 OBSERVANDO TENDÊNCIAS NA ENGENHARIA DE SOFTWARE

“Acredito poder haver um mercado mundial talvez para cinco computadores.”

Thomas Watson,
presidente da
IBM, 1943

PONTO-CHAVE

O “ciclo da excelência” apresenta uma visão realística da integração da tecnologia no curto prazo. No entanto, a tendência no longo prazo é exponencial.

A Seção 31.1 considerou brevemente as possibilidades fascinantes que podem surgir das tendências de longo prazo na computação e tecnologias relacionadas. E quanto ao curto prazo?

Barry Boehm [Boe08] sugere que “os engenheiros de software enfrentarão desafios muitas vezes enormes de tratar com rápidas mudanças, incerteza e emergência, confiabilidade, diversidade e interdependência, mas eles também têm oportunidades de fazer contribuições significativas”. Mas quais as tendências que nos possibilitarão o enfrentamento desses desafios nos próximos anos?

Na introdução deste capítulo, afirmei que “as tendências leves” têm um impacto significativo sobre a direção geral da engenharia de software. Mas outras tendências (“mais pesadas”) orientadas para a tecnologia de pesquisa permanecem importantes. As tendências nas pesquisas “são motivadas por percepções gerais do estado da arte e da prática, por percepções do pesquisador sobre as necessidades dos profissionais, por programas de fundos nacionais que buscam estratégias específicas e por puro interesse técnico” [Mil00a]. Tendências tecnológicas ocorrem quando pesquisas são extrapoladas para atender às necessidades da indústria e demandas do marketing.

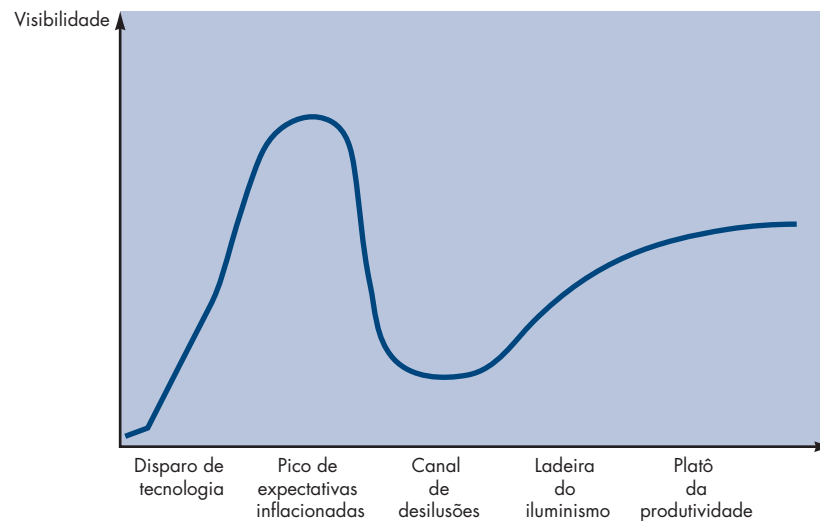
Na seção anterior, abordei o modelo da curva-S para a evolução da tecnologia. A curva-S é apropriada para considerar os efeitos de longo prazo das tecnologias básicas à medida que evoluem. Mas o que ocorre com as inovações, ferramentas e métodos mais modestos e de curto prazo? O Gartner Group [Gar08] — uma consultoria que estuda tendências da tecnologia em muitos ramos de atividade — desenvolveu um *ciclo da excelência para tecnologias emergentes*, representado na Figura 31.2. O ciclo Gartner Group apresenta cinco fases:

² Kurzweil [Kur05] apresenta um argumento técnico razoável que prevê forte tendência da inteligência artificial (que passará pelo teste de Turing) em 2029 e sugere que a evolução de humanos e máquinas começará a se combinar em 2045. A grande maioria dos leitores deste livro viverá para ver se isso de fato ocorrerá.

FIGURA 31.2

O ciclo da excelência do Gartner Group para tecnologias emergentes

Fonte: (Gar08)



- *Disparo da tecnologia* – avanço na pesquisa ou lançamento de um produto inovador que leva a grande cobertura da mídia e entusiasmo popular.
- *Pico de expectativas inflacionadas* – entusiasmo demasiado e projeções extremamente otimistas do impacto com base em um sucesso limitado, mas muito bem publicado.
- *Desilusão* – projeções demasiado otimistas do impacto não se confirmam, e os críticos começam a se manifestar; a tecnologia torna-se sem atrativos entre os conhecedores.
- *Ladeira do iluminismo* – utilização cada vez maior por uma ampla variedade de empresas leva a um melhor entendimento do verdadeiro potencial da tecnologia; aparecem métodos e ferramentas prontos para uso para suportar a tecnologia.
- *Platô de produtividade* – os benefícios do mundo real agora são óbvios, e o uso atinge uma porcentagem significativa do mercado em potencial.

Nem toda a tecnologia de engenharia de software segue esse caminho ao longo do ciclo da excelência. Em alguns casos, a decepção é justificada e a tecnologia fica relegada à obscuridade.

31.3 IDENTIFICANDO AS “TENDÊNCIAS LEVES”

“640K deve ser suficiente para todos.”

Bill Gates,
presidente da
Microsoft, 1981

Quais as tendências leves que afetarão as tecnologias relacionadas à engenharia de software?

Cada país com uma indústria de TI substancial tem um conjunto único de características que definem como os negócios são conduzidos, as dinâmicas organizacionais que surgem dentro de uma empresa, os aspectos de marketing que se aplicam a clientes locais e a cultura dominante que governa toda a interação humana. No entanto, algumas tendências em cada uma dessas áreas são universais e têm tanto a ver com sociologia, antropologia e psicologia de grupo (muitas vezes chamadas de “ciências leves”) quanto com pesquisa acadêmica ou industrial.

Conectividade e colaboração (possibilitadas pelas comunicações em banda larga) permitem a existência de equipes de software que não ocupam o mesmo espaço físico (teletrabalho e emprego de tempo parcial em um contexto local). Uma equipe colabora com outras que estão separadas por fuso horário, linguagem e cultura. A engenharia de software deve responder com um modelo de processo abrangente para “equipes distribuídas”, que seja ágil o bastante para atender às demandas da urgência, mas disciplinado o suficiente para coordenar diferentes grupos.

A *globalização* leva à força de trabalho diversificada (em termos de linguagem, cultura, solução de problemas, filosofia de administração, prioridades de comunicação e interação entre as pessoas). Isso, por sua vez, exige uma estrutura organizacional flexível. Equipes diferentes (em países diferentes) devem responder a problemas de engenharia de maneira que melhor atenda suas necessidades especiais, enquanto proporciona uniformidade para a execução de um proje-

to global. Esse tipo de organização sugere menos níveis de administração e ênfase maior na tomada de decisões em equipe. Pode levar a uma maior agilidade, mas apenas se mecanismos de comunicação tenham sido estabelecidos para que cada equipe possa entender o projeto e o status técnico (via conexão em rede) a qualquer tempo. Métodos e ferramentas da engenharia de software podem contribuir para a obtenção de algum nível de uniformidade (equipes falam a mesma “linguagem” implementada por métodos e ferramentas específicos). O processo de software pode proporcionar a estrutura para a existência desses métodos e ferramentas.

Em algumas regiões (nos Estados Unidos e na Europa, por exemplo), a população está envelhecendo. Essa inegável tendência demográfica (e cultural) indica que muitos engenheiros de software e gerentes experientes estarão deixando o campo de atividade durante a próxima década. A comunidade de engenharia de software deve responder com mecanismos viáveis que capturem o conhecimento desses gerentes e técnicos [por exemplo, o uso de *padrões* (Capítulo 12) é um passo na direção certa], para que fique à disposição das gerações futuras de profissionais. Em outras regiões do mundo, o número de jovens disponíveis para a indústria de software está explodindo. É a oportunidade de moldar uma cultura de engenharia de software sem o ônus de 50 anos de prejuízos da “velha escola”.

Estima-se que mais de um bilhão de novos consumidores entrarão no mercado mundial na próxima década. Os gastos dos consumidores nas “economias emergentes dobrarão, passando de 9 trilhões de dólares” [Pet06]. Não há dúvida de que uma porcentagem significativa desse gasto será com produtos e serviços que tenham um componente digital — que são baseados em software ou controlados por software. A implicação é uma demanda crescente por novos softwares. A questão então é, “Podem ser desenvolvidas novas tecnologias de engenharia de software para atender a essa demanda mundial?”. As tendências do mercado moderno muitas vezes são controladas pelo lado do fornecimento.³ Em outros casos, requisitos no lado da demanda controlam o mercado. Em qualquer das situações, um ciclo de inovação e demanda ocorre de maneira que muitas vezes torna-se difícil determinar quem vem primeiro!

Por fim, a própria cultura humana irá na direção da engenharia de software. Toda geração deixa sua própria marca na cultura local, e a nossa não será diferente. Faith Popcorn [Pop08], um conhecido consultor que se especializou em tendências culturais, caracteriza-as da seguinte maneira: “Nossas Tendências não são modismos. Nossas Tendências permanecem. Nossas Tendências evoluem. Elas representam forças escondidas, causas principais, necessidades humanas básicas, atitudes, aspirações. Elas nos ajudam a navegar pelo mundo, entender o que está acontecendo e por quê, e nos preparar para aquilo que ainda virá”. Uma discussão detalhada sobre como as tendências culturais modernas terão um impacto sobre a engenharia de software é mais bem apresentada por aqueles que se especializaram nas “ciências leves”.

31.3.1 Administrando a complexidade

Quando foi escrita a primeira edição deste livro (1982), produtos digitais da forma como conhecemos hoje não existiam, e os sistemas baseados em mainframe contendo um milhão de linhas de código-fonte (*lines of source code* — LOC) eram considerados muito grandes. Hoje, não é raro encontrar pequenos dispositivos digitais com 60 mil a 200 mil linhas de código-fonte de software personalizado, com alguns milhões de linhas de código para recursos do sistema operacional. Sistemas modernos baseados em computador contendo de 10 a 50 milhões de linhas de código não são incomuns.⁴ Em um futuro relativamente próximo, começarão a surgir sistemas⁵ que requerem mais de 1 bilhão de linhas de código.⁶

3 O lado fornecedor adota nos mercados uma abordagem do tipo “faça e eles comprarão”. Tecnologias especiais são criadas, e os consumidores se aglomeram para adotá-las — às vezes!

4 Os sistemas operacionais modernos dos PCs (por exemplo, Linux, MacOS e Windows) têm de 30 a 60 milhões de LOC. Sistemas operacionais de dispositivos móveis podem ter mais de 2 milhões de LOC.

5 Na realidade, esse “sistema” será um sistema de sistemas — centenas de aplicativos interoperáveis trabalhando juntos para atingir algum objetivo comum.

6 Nem todos os sistemas complexos são grandes. Um aplicativo relativamente pequeno (digamos, menos de 100 mil LOC) pode ainda ser bastante complexo.

Pense nisso por um instante!

Considere as interfaces para um sistema de 1 bilhão de linhas de código, para o mundo exterior, para outros sistemas interoperáveis, para a Internet (ou seu sucessor), e para os milhões de componentes internos que devem funcionar todos juntos para fazer esse monstro da computação operar corretamente. Há uma maneira confiável de garantir que todas essas conexões permitam que as informações fluam adequadamente?

Considere o próprio projeto. Como administramos o fluxo do trabalho e acompanhamos o progresso? As abordagens convencionais poderão ser escaladas muitas vezes em ordem de grandeza?

Considere o número de pessoas (e suas localizações) que estarão fazendo o trabalho, a ordenação dos profissionais e da tecnologia, o fluxo ininterrupto de alterações, a possibilidade de ambiente de sistema multiplataforma, multioperacional. Há uma maneira de administrar e coordenar indivíduos que estão trabalhando em um projeto enorme?

Considere o desafio da engenharia. Como podemos analisar dezenas de milhares de requisitos, limitações e restrições de uma forma que garanta que a inconsistência e ambiguidade, omissões e erros imediatamente sejam descobertos e corrigidos? Como podemos criar uma arquitetura de projeto que seja robusta o bastante para lidar com um sistema desse tamanho? Como os engenheiros de software poderão estabelecer um sistema de controle de alterações que terá de manipular centenas de milhares de alterações?

Considere o desafio da garantia da qualidade. Como podemos executar verificação e validação de modo significativo? Como você testa um sistema de 1 bilhão de LOC?

No passado, os engenheiros de software tentaram administrar a complexidade de uma forma que somente pode ser descrita como especial. Hoje, usamos processos, métodos e ferramentas para manter sob controle a complexidade. Mas o que acontecerá no futuro? A nossa abordagem atual estará à altura da tarefa a ser realizada?

“Não há razão para uma pessoa querer ter um computador em casa.”

Ken Olson,
presidente e
fundador da
Digital Equip-
ment Corp.,
1977

31.3.2 Software aberto

Conceitos como inteligência ambiente,⁷ aplicações ligadas a contexto, e computação invasiva/onipresente — todos focalizam a integração de sistemas baseados em software em um ambiente mais amplo do que um PC, um dispositivo de computação móvel, ou qualquer outro dispositivo digital. Essas visões separadas do futuro próximo da computação sugerem coletivamente o “software aberto” — software que é projetado para se adaptar a um ambiente em contínua mudança “auto-organizando sua estrutura e auto-adaptando seu comportamento” [Bar06].

Para ajudar a ilustrar os desafios que os engenheiros de software enfrentarão em um futuro previsível, considere a noção de *inteligência ambiente* (*ambient intelligence* — amI). Ducatel [Duc01] define a amI da seguinte maneira: “Pessoas estão rodeadas por interfaces inteligentes e intuitivas que estão embutidas em todos os tipos de objetos. O espaço da inteligência ambiente é capaz de reconhecer e responder à presença de diferentes indivíduos [enquanto trabalham] de uma maneira contínua e sem obstrução”.

Examinemos uma visão do futuro próximo na qual a amI se tornou onipresente. Você acaba de comprar um comunicador pessoal (chamado P-com, um dispositivo móvel de bolso) e passou as últimas semanas criando⁸ sua “imagem” — tudo, desde sua agenda diária, de coisas a fazer, livro de endereços, registros médicos, informações comerciais, documentos de viagem (coisas que você está procurando, por exemplo, um livro específico, um vinho raro, um curso local de arte em vidraria) e “Digital-Me” (D-Me) que o descreve com um nível de detalhes que possibilita uma apresentação digital aos outros (um tipo de *MySpace* ou *FaceBook* que anda com você). O P-com contém um identificador pessoal denominado “chave das chaves” — um identificador

⁷ Uma introdução muito boa e detalhada sobre inteligência ambiente pode ser encontrada no endereço www.emergingcommunication.com/volume6.html. Mais informações podem ser obtidas em www.ambientintelligence.org/.

⁸ Toda a interação com o P-com ocorre via comandos de reconhecimento de voz e instruções, que evoluíram tornando-se 99% precisos.

pessoal multifuncional que poderia fornecer acesso e habilitar consultas de uma grande variedade de dispositivos aml e sistemas.

É óbvio que entram em jogo aspectos significativos de privacidade e segurança. Um “sistema de gerenciamento de confiança — *trust management system*” [Duc01] será parte integral da aml e controlará os privilégios que possibilitam a comunicação com sistemas de redes, saúde, entretenimento, finanças, emprego e pessoal.

Novos sistemas com capacidade aml serão acrescentados à rede constantemente, cada um deles proporcionando recursos úteis e demandando acesso ao seu P-com. Portanto, o software P-com deve ser projetado de forma que possa se adaptar aos requisitos que surgem sempre que novos sistemas aml estejam on-line. Há muitas maneiras de fazer isso, mas resumindo: o P-com deve ser flexível e robusto em aspectos que o software convencional não consegue atingir.

31.3.3 Requisitos emergentes

No início de um projeto de software, há uma autenticidade que se aplica igualmente a todos os envolvidos: “Você não sabe o que não sabe”. Isso significa que os clientes raramente definem requisitos “estáveis”. Indica também que os engenheiros de software não podem prever onde ocorrerão as ambiguidades e inconsistências. Os requisitos mudam — mas isso não é novidade.

Na medida em que os sistemas se tornam mais complexos, mesmo uma tentativa rudimentar de definir requisitos abrangentes está destinada ao fracasso. Uma definição de objetivos globais pode ser possível, pode ser conseguido um esboço dos objetivos intermediários, mas requisitos estáveis — sem chance! Os requisitos surgirão conforme todos os envolvidos na engenharia e construção de um sistema complexo aprenderem mais sobre esse sistema, sobre o ambiente no qual ele reside e sobre os usuários que vão interagir.

Essa realidade implica uma série de tendências de engenharia de software. Primeiro, devem ser desenhados modelos de processo para aderir à mudança e adotar as crenças básicas da filosofia ágil (Capítulo 3). Em seguida, métodos que resultem em modelos elaborados (por exemplo, modelos de requisito e de desenho) devem ser usados criteriosamente, porque esses modelos mudarão repetidamente à medida que for adquirido mais conhecimento sobre o sistema. Por fim, ferramentas que suportem tanto os processos quanto os métodos devem facilitar a adaptação e mudança.

Mas há outro aspecto dos requisitos emergentes. Para a grande maioria do software desenvolvido até hoje, a fronteira entre o sistema baseado em software e seu ambiente externo é estável. A fronteira pode mudar, mas isso ocorrerá de uma maneira controlada, permitindo que o software seja adaptado como parte de um ciclo de manutenção. Essa opinião está começando a mudar. O software aberto (*open-world software*) (Seção 31.2.2) exige que os sistemas baseados em computador “se adaptem e reajam às mudanças dinamicamente, mesmo que sejam imprevisíveis” [Bar06].

Pela sua natureza, os requisitos emergentes levam à mudança. Como controlamos a evolução de um aplicativo ou sistema amplamente usado durante toda a sua vida útil, e que efeito isso tem sobre a maneira como projetamos software?

Conforme o número de alterações aumenta, a possibilidade de efeitos colaterais não desejados também aumenta. Isso deverá ser um motivo de preocupação quando sistemas complexos com requisitos emergentes se tornarem comuns. A comunidade de engenharia de software deve desenvolver métodos que ajudem as equipes a prever o impacto das mudanças no sistema inteiro, moderando assim efeitos colaterais indesejados. Hoje, nossa habilidade para tanto é severamente limitada.

31.3.4 O mix de talentos

A natureza de uma equipe de engenharia de software pode mudar à medida que os sistemas baseados em software fiquem mais complexos, as comunicações e colaboração entre equipes globais transformarem-se em lugar-comum e os requisitos emergentes (com o fluxo de mu-



Em virtude de os requisitos emergentes já serem uma realidade, sua organização deverá pensar em adotar um modelo de processo incremental.

danças resultantes) se tornarem a norma. Cada equipe de software deve contribuir com talento criativo e habilidades técnicas para sua parte de um sistema complexo, e o processo todo deve permitir que o resultado dessas ilhas de talento se combine efetivamente.

Alexandra Weber Morales [Mor05] sugere o mix de talentos de uma “equipe ideal de software” (*software dream team*). O *Cérebro* é o arquiteto principal capaz de lidar com as demandas dos interessados e mapeá-las em uma estrutura de tecnologia extensível e implementável. O *Data Grrl* é o guru de base de dados e estrutura de dados que “ataca vigorosamente através de linhas e colunas com profundo conhecimento da lógica de predicados e teoria dos conjuntos no que se refere ao modelo relacional”. O *Blocker* é um líder técnico (gerente) que permite que a equipe trabalhe livre de interferências enquanto ao mesmo tempo garante que a colaboração está ocorrendo. O *Hacker* é um programador perfeito que está à vontade com padrões e linguagens e pode usar ambas eficazmente. O *Gatherer* “descobre habilmente requisitos de sistema com visão antropológica” e os expressa com clareza.

31.3.5 Blocos básicos de software

“A resposta criativa para a tecnologia digital é adotá-la como uma nova janela sobre tudo o que é eternamente humano, e usá-la com paixão, sabedoria, bravura e alegria.”

Ralph Lombreglia

Todos que já adotamos uma filosofia de engenharia de software já enfatizamos a necessidade de reutilização — de código-fonte, classes orientadas a objeto, componentes, padrões e software de prateleira. Embora a comunidade de engenharia tenha feito progresso na sua tentativa de capturar conhecimento e reutilizar soluções aprovadas, uma porcentagem significativa do software atual continua a ser criada “desde o início”. Parte da razão de tal procedimento é um contínuo desejo (por parte dos interessados e profissionais de engenharia de software) de “soluções únicas”.

No mundo do hardware, os OEMs (*original equipment manufactures*) dos dispositivos digitais usam produtos-padrão específicos de aplicativo (*application-specific standard products* — ASSPs) produzidos por fabricantes de circuitos integrados quase exclusivamente. Esse “mercador de hardware” fornece os blocos básicos necessários para implementar tudo, desde um telefone móvel até um HD-DVD player. Cada vez mais, os mesmos OEMs usam o “mercador de software” — blocos básicos de software projetados especificamente para um domínio de aplicação único [por exemplo, dispositivos VoIP]. Michael Ward [War07] comenta:

Uma vantagem do uso de componentes de software é que o OEM pode alavancar a funcionalidade proporcionada pelo software sem ter de desenvolver especialidades internas nas funções específicas ou investir tempo de desenvolvedor no trabalho de implementar e validar os componentes. Outras vantagens incluem a habilidade para adquirir e fornecer apenas o conjunto específico de funcionalidades necessárias ao sistema, bem como a habilidade para integrar esses componentes em uma arquitetura já existente.

No entanto, a abordagem de componente de software tem uma desvantagem porque há certo nível de esforços necessários para integrar os componentes individuais no produto global. Esse desafio pode ser ainda mais complicado se os componentes originam-se de uma variedade de fornecedores, cada um com suas próprias metodologias de interface. Conforme outras fontes de componentes forem usadas, aumenta o trabalho de administrar um número maior de fornecedores, e há um risco maior de se encontrarem problemas com a interação através de componentes de diferentes origens.

Além dos componentes agregados como um mercado de software, há uma tendência crescente em adotar *soluções de plataforma de software* que “incorporam coleções de funcionalidades relacionadas, fornecidas tipicamente em uma estrutura de software integrada” [War07]. Uma plataforma de software libera um OEM do trabalho associado ao desenvolvimento de funcionalidade básica e, em vez disso, permite que o OEM dedique trabalho de software para aquelas características que diferenciam seu produto.

31.3.6 Mudando as percepções de “valor”

Durante os últimos 25 anos do século 20, a pergunta operativa que os homens de negócios faziam ao discutirem software era: “Por que custa tão caro?”. Essa pergunta raramente é feita

nos dias de hoje e ela foi substituída por outra: “Por que não podemos obter esse (software e/ou produto baseado em software) mais rapidamente?”.

Quando se considera software para computador, a percepção moderna está mudando de valor nos negócios (custo e lucratividade) para valores de clientes que incluem rapidez na entrega, riqueza de funcionalidade e qualidade geral do produto.

31.3.7 Código aberto

Quem é o proprietário do software que você ou sua organização utiliza? Cada vez mais, a resposta é “todos”. O movimento “código aberto” (*open source*) tem sido descrito da seguinte maneira [OSO08]: “Código aberto é um método de desenvolvimento de software que utiliza o poder da revisão por pares (*peer review*) distribuída e a transparência do processo. A premissa do código aberto é melhor qualidade, maior confiabilidade, maior flexibilidade, menor custo e o fim do aprisionamento tecnológico (*vendor lock-in*) predatório”. O termo *código aberto*, quando aplicado a software de computador, implica que os produtos de engenharia de software (modelos, código-fonte, conjuntos de teste) são abertos ao público e podem ser revistos e ampliados (com controles) por qualquer um com interesse e permissão.

Uma “equipe” de código aberto tem um conjunto de membros, em tempo integral, do tipo “time dos sonhos” (Seção 31.3.4); o número de pessoas trabalhando no software aumenta e diminui conforme a necessidade da aplicação. O poder da equipe de código aberto é derivado da constante revisão por pares e refatoração (*refactoring*) de projeto/código que resulta em uma lenta progressão em direção a uma solução ótima.

Se você tiver mais interesse, Weber [Web05] fornece uma introdução valiosa, e Feller e seus colegas [Fel07] editaram uma antologia abrangente e objetiva que considera os benefícios e problemas associados a código aberto.



Tecnologias a ser observadas

Muitas tecnologias emergentes podem ter um impacto significativo sobre os tipos de sistemas baseados em computadores que evoluem. Elas se somam aos desafios que confrontam os engenheiros de software. Vale notar as seguintes tecnologias:

Grid computing – Esta tecnologia (disponível hoje) cria uma rede que aproveita os bilhões de ciclos de CPU não utilizados de cada máquina na rede e permite que a computação de trabalhos extremamente complexos seja feita sem precisar de um supercomputador dedicado. Para um exemplo abrangendo mais de 4,5 milhões de computadores, visite o site <http://setiathome.berkeley.edu/>.

Computação aberta – “Ambiente, implícito, invisível e adaptável. Situação em que os dispositivos de rede agregados no ambiente proporcionam conectividade e serviços que passam despercebidos todo o tempo” [McC05].

Microcomércio – Novo ramo do comércio eletrônico que cobra quantias muito pequenas para o acesso ou compra de várias formas de propriedade intelectual. O iTunes da Apple é um exemplo bastante usado.

Máquinas cognitivas – O “santo graal” do campo da robótica é desenvolver máquinas que tenham ciência de seu ambiente, que possam “captar informações, responder a situações em contínua mudança e interagir com pessoas naturalmente” [PCM03]. As máquinas cognitivas ainda

estão nos primeiros estágios de desenvolvimento, mas o potencial é enorme.

Monitores OLED – O monitor OLED (*OLED display*) “usa uma molécula traçadora à base de carbono que emite luz quando uma corrente elétrica passa através dela. Junte grandes quantidades dessas moléculas e você terá um monitor superfino com uma qualidade assombrosa — sem nenhuma luz de fundo consumindo potência” [PCM03]. O resultado — monitores ultrafinos que podem ser enrolados ou dobrados, posicionados sobre uma superfície curva, ou adaptado de alguma outra forma a um ambiente específico.

RFIDs – Os identificadores por radiofrequência (*radio frequency identification*) trazem a computação aberta para uma base industrial e para o ramo de produtos de consumo. Qualquer coisa, de tubos de pasta de dente a motores de automóveis, pode ser identificada quando se movimenta através de uma esteira até seu destino final.

Web 2.0 – Amplo arranjo de serviços da Web que levará a uma integração ainda maior da Internet tanto no comércio quanto na computação pessoal.

Para mais discussões sobre tecnologias apresentadas em uma combinação especial de vídeo e material impresso, visite o site da Consumer Electronics Association, www.ce.org/Press/CEA_Pubs/135.asp.

INFORMAÇÕES

31.4 DIREÇÕES DA TECNOLOGIA

“Mas para que serve?”

Engenheiro da divisão de computação avançada da IBM, 1968, comentando sobre um microchip

Sempre pensamos que a engenharia de software mudará mais rapidamente do que está ocorrendo. Uma nova tecnologia “excelente” (poderia ser um novo processo, um método especial ou uma ferramenta interessante) é introduzida, e os críticos sugerem que “tudo” mudará. Mas a engenharia de software é muito mais do que tecnologia — ela trata de pessoas e suas habilidades em comunicar necessidades e inovar para tornar aquelas necessidades uma realidade. Sempre que há pessoas envolvidas, as mudanças ocorrem lentamente de maneira impulsiva e irregular. Apenas quando um “ponto de virada” [Gla02] é atingido, que uma tecnologia se propaga através da comunidade de engenharia de software e uma ampla mudança realmente ocorre.

Nesta seção, examinaremos algumas tendências em processos, métodos e ferramentas que podem ter alguma influência sobre a engenharia de software durante a próxima década. Elas conduzirão a um ponto de virada? Temos de esperar para ver.

31.4.1 Tendências de processo

Podemos dizer que todas as tendências de negócios, organizacionais e culturais discutidas na Seção 31.3 reforçam a necessidade de processo. Mas as estruturas abordadas no Capítulo 30 fornecem um roteiro para o futuro? As estruturas de processo irão evoluir para buscar um melhor equilíbrio entre a disciplina e a criatividade? Os processos de software se adaptarão às diferentes necessidades dos interessados que solicitam o software, aqueles que o criam e aqueles que o utilizam? O software pode proporcionar um meio de reduzir o risco dos três componentes ao mesmo tempo?

Essas e outras questões permanecem pendentes. No entanto, algumas tendências começam a surgir. Conradi e Fuggetta [Con02] sugerem seis “teses sobre como aperfeiçoar e melhor aplicar as estruturas SPI”. Eles iniciam a discussão com a seguinte afirmação:

A meta de um promovedor de software é selecionar o melhor prestador de serviço objetivamente e racionalmente. A meta de uma empresa de software é sobreviver e crescer em um mercado competitivo. A meta de um usuário final é adquirir o produto que pode resolver o problema certo, na hora certa, a um preço aceitável. Não podemos esperar que uma mesma abordagem e consequente esforço de SPI possa acomodar todos esses diferentes pontos de vista.

Nos próximos parágrafos, adaptarei as teses propostas por Conradi e Fuggetta [Con02] para sugerir possíveis tendências de processo durante a próxima década.

1. *À medida que as estruturas SPI evoluem, darão ênfase a “estratégias que focalizam a orientação e inovação de produto”* [Con02]. Em um mundo de rápidas mudanças no desenvolvimento de software, estratégias SPI de longo prazo raramente sobrevivem em um ambiente de negócios dinâmicos. Há muitas mudanças ocorrendo rapidamente. Isso significa que um roteiro estável, passo a passo para SPI, pode precisar ser substituído por uma estrutura que enfatize os objetivos de curto prazo com uma orientação para produto. Se os requisitos para uma nova linha de produto baseado em software surgirem no decorrer de uma série de versões de produto incrementais (a ser fornecidas ao usuário final via Web), a organização de software pode precisar melhorar sua habilidade em administrar a mudança. Melhoras de processo associadas a gerenciamento de mudança devem ser coordenadas com o ciclo de versões do produto, para que melhore o gerenciamento de mudanças e ao mesmo tempo não seja confuso.
2. *Pelo fato de os engenheiros de software terem uma boa noção do ponto frágil do processo, as mudanças em geral deverão ser motivadas por suas necessidades e começar de baixo para cima.* Conradi e Fuggetta [Con02] sugerem que as atividades de SPI no futuro deverão “usar um planar simples e focalizado para começar, não uma ampla avaliação”. Concentrando-se nos esforços de SPI de forma restrita e trabalhando de baixo para cima, os profissionais começarão logo a ver mudanças substanciais em como é conduzido o trabalho de engenharia de software.

 Quais são as tendências de processo mais prováveis na próxima década?

3. *A tecnologia de processo automatizado de software (automated software process technology — SPT) se distanciará do gerenciamento global de processo (suporte com base ampla de todo o processo de software) para concentrar-se nos aspectos que podem se beneficiar mais da automação.* Ninguém é contra ferramentas e automação, mas em muitas situações, a SPT não atingiu seu objetivo (veja a Seção 31.2). Para maior eficácia, ela deverá focalizar atividades de apoio (Capítulo 2) — os elementos mais estáveis do processo de software.
4. *Haverá mais ênfase ao retorno sobre o investimento das atividades de SPI.* No Capítulo 30, você aprendeu que o retorno sobre o investimento (*return on investment* — ROI) pode ser definido como:

$$\text{ROI} = \left[\frac{\Sigma(\text{benefícios}) - \Sigma(\text{custos})}{\Sigma(\text{custos})} \right] \times 100\%$$

Até hoje, as organizações de software têm se empenhado para delinear claramente os “benefícios” de maneira quantitativa. Pode-se afirmar [Con02] que “precisamos, portanto, de um modelo padronizado de valor de mercado, como o empregado na Cocomo II (veja o Capítulo 26) para levar em conta as iniciativas de melhorias de software”.

5. *À medida que o tempo passa, a comunidade do software pode vir a entender que a especialização em sociologia e antropologia pode ter tanto ou muito mais a ver com uma SPI bem-sucedida, quanto outras disciplinas, mais técnicas.* Além disso, a SPI muda a cultura organizacional, e a mudança cultural envolve indivíduos e grupos de profissionais. Conradi e Fuggetta [Con02] observam corretamente que “os desenvolvedores de software são trabalhadores de conhecimento. Tendem a responder negativamente a ordem de cima sobre como fazer o trabalho ou como mudar processos”. Pode-se aprender muito examinando a sociologia de grupos para melhor entender maneiras eficazes de introduzir uma mudança.
6. *Novos modos de aprender podem facilitar a transição para um processo de software mais eficaz.* Nesse contexto, “aprender” implica aprender com sucessos e erros. Uma organização de software que coleta métricas (Capítulos 23 e 25) permite a si mesma entender como os elementos de um processo afetam a qualidade do produto final.

31.4.2 O grande desafio

Há uma tendência inegável — os sistemas baseados em software sem dúvida se tornarão maiores e mais complexos com o passar do tempo. É a engenharia desses sistemas grandes e complexos, independentemente da plataforma ou domínio de aplicação, que apresenta o “grande desafio” [Bro06] para os engenheiros de software. Manfred Broy [Bro06] afirma que os engenheiros de software podem enfrentar “o desafio assustador do desenvolvimento de sistemas complexos de software”, criando novas abordagens para entender modelos de sistema e usando esses modelos como base para a construção de software de alta qualidade da nova geração.

Conforme a comunidade de engenharia de software desenvolve novas abordagens motivadas por modelo (assunto discutido mais adiante nesta seção) para a representação de requisitos de sistema e projeto, devem ser tratadas as seguintes características [Bro06]:

 **Que características de sistema os analistas e projetistas devem considerar para futuras aplicações?**

- *Multifuncionalidade* – Depois que os dispositivos digitais evoluíram para sua segunda e terceira geração, começaram a apresentar um amplo conjunto de funções às vezes não relacionadas. O telefone móvel, considerado um dispositivo de comunicação, agora é usado para fotos, calendário, navegação e como reproduzidor de música. Se as interfaces abertas vieram para ficar, esses dispositivos móveis serão usados para muitas outras funções nos próximos anos. Conforme observa Broy [Bro06], “os engenheiros devem descrever o contexto detalhado no qual as funções serão fornecidas e, mais importante, devem identificar as interações potencialmente perigosas entre diferentes características do sistema”.
- *Reatividade e temporização (timeliness)* – os dispositivos digitais interagem cada vez mais com o mundo real e devem reagir aos estímulos externos no tempo. Eles devem estabe-

lecer interface com um amplo conjunto de sensores e devem responder em um intervalo de tempo apropriado para a tarefa em questão. Devem ser desenvolvidos novos métodos que (1) ajudem os engenheiros de software a prever a cadência das várias características reativas e (2) implementem características menos dependentes da máquina e mais portáteis.

- *Novos modos de interação de usuário* – o teclado e o mouse funcionam muito bem em um ambiente de PC, mas as tendências abertas para software significam que novos modos de interação devem ser modelados e implementados. Independentemente do fato de essas novas abordagens usarem interfaces multitoque, de reconhecimento de voz ou interfaces mentais diretas,⁹ as novas gerações de software para dispositivos digitais devem modelar as novas interfaces homem-computador.
- *Arquiteturas complexas* – um automóvel de luxo tem mais de 2 mil funções controladas por software que reside em uma arquitetura de hardware complexa, incluindo múltiplas CPUs, estrutura de barramento sofisticada, atuadores, sensores e uma interface humana cada vez mais sofisticada, e muitos componentes relacionados com a segurança. Sistemas ainda mais complexos estão no horizonte próximo, apresentando desafios significativos para os projetistas de software.
- *Sistemas heterogêneos distribuídos* – os componentes de tempo real de qualquer sistema embutido moderno podem ser conectados através de um barramento interno, uma rede sem fio ou da Internet (ou as três coisas).
- *Criticidade* – o software tornou-se o componente pivô em todos os sistemas críticos nos negócios e em muitos sistemas em termos de segurança. Contudo, a comunidade de engenharia de software apenas começou a aplicar os princípios mais básicos de segurança de software.
- *Variabilidade de manutenção* – a vida do software em um dispositivo digital raramente dura além de 3 a 5 anos, mas os sistemas complexos de aviãoica instalados em uma aeronave têm uma vida útil de pelo menos 20 anos. O software dos automóveis fica a meio-termo. Isso deverá ter um impacto sobre o projeto?

Broy [Bro06] afirma que essas e outras características do software podem ser administradas somente se a comunidade desenvolver uma filosofia de engenharia de software distribuída e colaborativa mais eficaz, melhores abordagens de requisitos de engenharia, uma abordagem mais robusta do desenvolvimento motivado por modelo e melhores ferramentas de software. Nas próximas seções, exploraremos rapidamente cada uma dessas áreas.

31.4.3 Desenvolvimento colaborativo

Parece muito óbvio, mas direi mesmo assim: *a engenharia de software é uma tecnologia de informação*. Desde o princípio de qualquer projeto de software, todos os interessados devem compartilhar informações — sobre metas e objetivos básicos, sobre requisitos básicos de sistema, sobre problemas de projeto de arquitetura, sobre quase todos os aspectos do software a ser criado.

Hoje, os engenheiros de software colaboram em diferentes fusos horários e diferentes países, e todos eles devem compartilhar informações. O mesmo vale para projetos abertos nos quais centenas ou milhares de desenvolvedores de software trabalham para criar uma aplicação aberta. Uma vez mais, as informações devem ser disseminadas para que possa ocorrer a colaboração aberta.

O desafio para a próxima década é desenvolver métodos e ferramentas que facilitem essa colaboração. Hoje, continuamos nos esforçando para facilitar a colaboração. Eugene Kim [Kim04] comenta:

PONTO-CHAVE

A colaboração envolve a disseminação ao longo do tempo e um processo eficaz para comunicação e criação de projeto.

⁹ Uma breve discussão sobre as interfaces mentais diretas pode ser encontrada em http://en.wikipedia.org/wiki/Brain_computer_interface, e um exemplo comercial é descrito no site <http://au.gamespot.com/news/6166959.html>.

Considere uma tarefa colaborativa básica: compartilhamento de documento. Muitos aplicativos (tanto comerciais quanto abertos) prometem resolver o problema do compartilhamento de documento e, ainda mais, o método predominante para compartilhar arquivos é enviá-los por e-mail de um lado para outro. Esse é o equivalente computacional de transferência eletrônica de informações. Se as ferramentas que pretendem resolver esse problema são boas, por que não utilizá-las?

Vemos problemas similares em outras áreas básicas. Posso entrar em qualquer reunião em qualquer lugar do mundo com uma folha de papel na mão, e posso estar seguro de que todas as pessoas poderão ler, fazer marcações, passar adiante e arquivar. Não posso dizer o mesmo dos documentos eletrônicos. Não posso fazer anotações em uma página da Web ou usar o mesmo sistema de arquivamento para meus e-mails e meus documentos do Word, pelo menos não de uma maneira que seja interoperável com aplicações em minha própria máquina e nas máquinas de outras pessoas. Por que não?

...Para causar um impacto real no espaço colaborativo, as ferramentas precisam não apenas ser boas, mas devem ser interoperáveis.

Mas a falta de ferramentas colaborativas abrangentes é apenas uma parte do desafio enfrentado por aqueles que devem desenvolver software de forma colaborativa.

Hoje, uma porcentagem significativa¹⁰ dos projetos de TI são terceirizados internacionalmente, e o número crescerá substancialmente durante a próxima década. Não é surpresa que Bhat e seus colegas [Bha06] afirmam que a engenharia de requisitos é a atividade crucial em um projeto de terceirização. Eles identificam um conjunto de fatores de sucesso que levam a esforços colaborativos bem-sucedidos:

- *Metas compartilhadas* – as metas de projeto devem ser claramente enunciadas, e todos os interessados devem entendê-las e concordar com seu intento.
- *Cultura compartilhada* – diferenças culturais devem ser claramente definidas, e deve ser desenvolvida uma abordagem educacional (que ajudará a moderar essas diferenças) e uma abordagem de comunicação (que facilitará a transferência de conhecimentos).
- *Processo compartilhado* – de alguma maneira, o processo serve de esqueleto de um projeto colaborativo, proporcionando um meio uniforme para avaliar o progresso e a direção e introduzir uma “linguagem” técnica comum para todos os membros da equipe.
- *Responsabilidade compartilhada* – todo membro de equipe deve reconhecer a importância da engenharia de requisitos e trabalhar para proporcionar a melhor definição possível do sistema.

Quando combinados, esses fatores de sucesso levam à “confiança” — uma equipe global que pode confiar em grupos diversos para executar a tarefa que lhes foi atribuída.

31.4.4 Engenharia de requisitos

As ações básicas de engenharia de requisitos (*requirements engineering* — RE) — evocação, elaboração, negociação, especificação e validação — foram apresentadas nos Capítulos 5 a 7. O sucesso ou fracasso dessas ações têm forte influência sobre o sucesso ou fracasso do processo inteiro de engenharia de software. Contudo, a RE tem sido comparada com “tentar colocar uma braçadeira de mangueira” [Gon04]. Conforme já notamos em várias partes deste livro, os requisitos de software tendem a continuar mudando e, com o surgimento dos sistemas abertos, requisitos emergentes (e mudanças quase contínuas) podem se tornar coisa normal.

Hoje, muitas abordagens “informais” de RE começam com a criação de cenários de usuário (por exemplo, casos de uso). Abordagens mais informais criam um ou mais modelos de requisitos e os utilizam como base para o projeto. Métodos formais permitem a um engenheiro de software representar requisitos usando uma notação matemática que pode ser verificada. Tudo pode funcionar razoavelmente bem quando os requisitos são estáveis, mas não resolve imediatamente o problema dos requisitos dinâmicos ou emergentes.

¹⁰ Aproximadamente 20% de um orçamento típico de TI para grandes empresas é dedicado atualmente à terceirização, e essa porcentagem está aumentando a cada ano. (Fonte: www.logicacmg.com/page/400002849.)

PONTO-CHAVE

“Novos subprocessos de RE incluem: (1) melhor aquisição de conhecimento, (2) iteração ainda melhor, e (3) ferramentas mais eficazes de comunicação e coordenação.”

Há uma série de direções distintas de pesquisa de engenharia de requisitos, incluindo processamento de linguagem natural com base em descrições textuais traduzidas para representações mais estruturadas, uma maior confiabilidade em bases de dados para estruturação e entendimento de requisitos de software, o uso de padrões de RE para descrever problemas típicos e soluções quando são executadas as tarefas de engenharia de requisitos e engenharia de requisitos orientada para metas. No entanto, no nível industrial, ações de RE permanecem relativamente informais e surpreendentemente básicas. Para melhorar a maneira como os requisitos são definidos, a comunidade de engenharia de software provavelmente implementará três subprocessos distintos enquanto a RE é executada [Gli07]: (1) melhora na aquisição de conhecimentos e compartilhamento de conhecimentos que possibilita entendimento mais completo das restrições do domínio de aplicação e necessidades dos interessados, (2) maior ênfase sobre a iteração quando os requisitos são definidos, e (3) ferramentas mais eficazes de comunicação e coordenação que permitem a todos os interessados colaborar eficazmente.

Os subprocessos de RE descritos no parágrafo anterior só terão sucesso se forem integrados adequadamente em uma abordagem evolutiva da engenharia de software. Como a solução de problemas baseados em padrões e as soluções baseadas em componentes começam a conquistar muitos domínios de aplicação, a RE deve conciliar o desejo de agilidade (entrega incremental rápida) e os requisitos emergentes inerentes resultantes. A natureza concorrente de muitos modelos de engenharia de software significa que a RE será integrada a atividades de projeto e construção. Como consequência, a noção de uma “especificação de software” estática está começando a desaparecer, para ser substituída por “requisitos motivados por valor” [Som05] derivados quando os interessados respondem às características e funções fornecidas nos primeiros incrementos de software.

31.4.5 Desenvolvimento de software controlado por modelo

Os engenheiros de software lidam com abstração em quase todas as etapas no processo de engenharia de software. Quando o projeto começa, as abstrações em nível de arquitetura e de componente são representadas e julgadas. Elas devem então ser traduzidas para uma representação de linguagem de programação que transforma o projeto (uma abstração de nível relativamente alto) em um sistema operável com um ambiente de computação específico (baixo nível de abstração). O *desenvolvimento de software controlado por modelo*¹¹ acopla linguagens de modelagem específicas de domínio com mecanismo de transformação e geradores para facilitar a representação da abstração em altos níveis e, então, a transforma em níveis mais baixos [Sch06].

Linguagens de modelagem específicas de domínio (domain-specific modeling languages – DSMLs) representam “estrutura de aplicativo, comportamento e requisitos dentro de domínios de aplicação particulares” e são descritas com metamodelos que “definem as relações entre conceitos no domínio e especificam precisamente as semânticas fundamentais e as restrições associadas a esses conceitos de domínio” [Sch06]. A principal diferença entre uma DSML e uma linguagem de modelagem de uso geral como a UML (Apêndice 1) é que a DSML está sintonizada com conceitos de projeto inerentes ao domínio de aplicação e pode, portanto, representar relações e restrições entre elementos de projeto de modo eficiente.

31.4.6 Projeto pós-moderno

Em um artigo interessante sobre projeto de software na “era pós-moderna”, Philippe Kruchten [Kru05] faz a seguinte observação:

A ciência da computação não atingiu ainda o contexto geral que consiga explicar sua totalidade — ainda não descobrimos as leis fundamentais do software que teriam o mesmo papel das leis fundamentais da física em outras disciplinas de engenharia. Ainda sentimos o gosto amargo do estouro da bolha da Internet e do pesadelo Y2K. Assim, nesta era pós-moderna, em que parece que tudo importa um pouco, embora quase nada importe, quais são os próximos rumos para o projeto de software?

PONTO-CHAVE

Abordagens controladas por modelo tratam de um desafio contínuo para todos os desenvolvedores de software — como representar o software em um nível mais alto de abstração do que o código.

11 O termo *model-driven engineering* (MDE) também é usado.

Parte de qualquer tentativa de entender tendências em projeto de software é estabelecer fronteiras para o projeto. Onde termina a engenharia de requisitos e começa o projeto? Onde termina o projeto e começa a geração de código? A resposta a essas questões não é tão fácil como pode parecer. Embora o modelo de requisitos devesse concentrar-se “no que”, não em “como”, todo analista faz um pouco de projeto e quase todos os projetistas fazem um pouco de análise. De forma semelhante, conforme o projeto de componentes de software se aproxima do detalhe algorítmico, um projetista começa a representar o componente em um nível de abstração que se aproxima do código.

O projeto pós-moderno continuará a enfatizar a importância da arquitetura de software (Capítulo 9). O projetista deve definir um problema arquitetônico, tomar uma decisão relacionada com esse problema e então definir claramente as suposições, restrições e implicações que a decisão impõe ao software como um todo. Mas há uma estrutura na qual os problemas podem ser descritos e a arquitetura pode ser definida? O desenvolvimento de software orientado a aspecto (Capítulo 2) ou desenvolvimento de software controlado por modelo (Seção 31.4.4) podem se tornar abordagens de projeto importantes nos próximos anos, mas ainda é muito cedo para dizer. Pode ocorrer de grandes avanços em desenvolvimento baseado em componente (Capítulo 10) levar a uma filosofia de projeto que enfatiza a montagem de componentes existentes. Se o passado é um prólogo, é muito provável que muitos métodos “novos” de projeto surgirão, mas poucos percorrerão a curva da excelência (Figura 31.2) muito além do “canal da desilusão”.

31.4.7 Desenvolvimento baseado em teste

Os requisitos controlam o projeto, e o projeto estabelece uma fundação para a construção. Essa simples realidade de engenharia de software funciona razoavelmente bem e é essencial quando se cria uma arquitetura de software. No entanto, uma mudança sutil pode trazer benefícios significativos quando são considerados projeto e construção em nível de componente.

Em *desenvolvimento baseado em teste* (*test-driven development* — TDD), requisitos para um componente de software servem de base para a criação de uma série de casos de teste que exercitam a interface e tentam encontrar erros nas estruturas de dados e funcionalidade fornecida pelo componente. A TDD não é realmente uma nova tecnologia, mas sim uma tendência que enfatiza o projeto de casos de teste *antes* da criação do código-fonte.¹²

O processo TDD segue o fluxo procedural simples ilustrado na Figura 31.3. Antes de ser criado o primeiro segmento de código, um engenheiro de software cria um teste para exercitar o código (tentando fazer o código falhar). É então escrito o código para satisfazer ao teste. Se passar, um novo teste é criado para o próximo segmento de código a ser desenvolvido. O processo continua até que o componente esteja completamente codificado e todos os testes executam sem erro. No entanto, se algum teste conseguir descobrir um erro, o código existente é refeito (corrigido) e todos os testes criados até aquele ponto são executados novamente. Esse fluxo iterativo continua até que não haja mais teste a ser criado, implicando que o componente satisfaz a todos os requisitos definidos para ele.

Durante o TDD, o código é desenvolvido em incrementos muito pequenos (uma subfunção de cada vez), e nenhum código é escrito enquanto não houver um teste para experimentá-lo. Você deve observar que cada iteração resulta em um ou mais novos testes que são acrescentados a um conjunto de testes de regressão que roda com cada mudança. Isso é feito para garantir que o novo código não tenha gerado efeitos colaterais que causam erros no código anterior.

Em TDD, testes controlam o projeto detalhado de componente e o código-fonte resultante. Os resultados desses testes causam modificações imediatas no projeto do componente (via código), e o que é mais importante, o componente resultante (quando completado) foi verificado em condição isolada (*stand-alone*). Se você tiver mais interesse em TDD, veja [Bec04b] ou [Ast04].

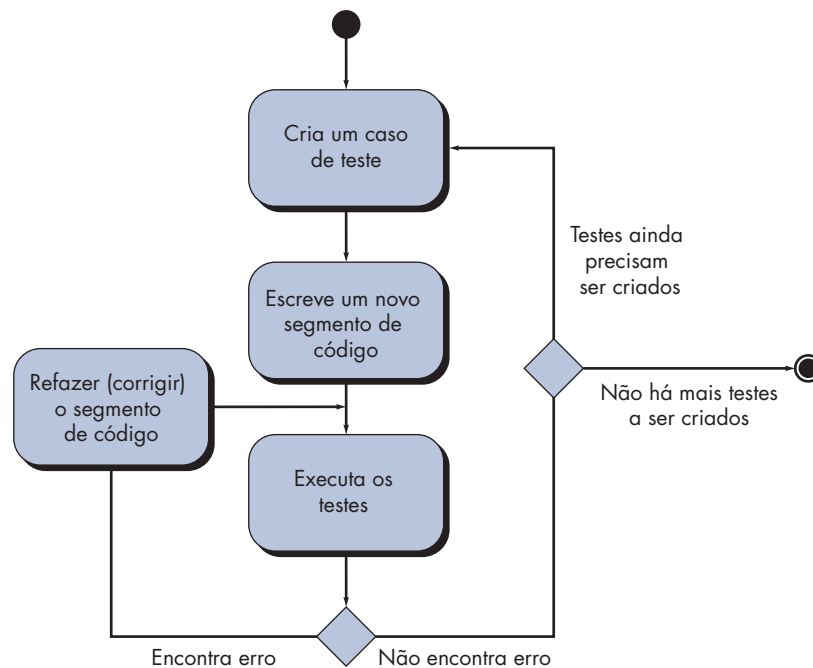
PONTO-CHAVE

“TDD é uma tendência que enfatiza o projeto de casos de teste *antes* da criação de código-fonte”.

12 Recorde-se que a *Extreme Programming* (Capítulo 3) enfatiza essa abordagem como parte de seu modelo de processo ágil.

FIGURA 31.3

O fluxo de desenvolvimento controlado por teste



31.5 TENDÊNCIAS RELACIONADAS COM FERRAMENTAS

Centenas de ferramentas de engenharia de software de nível industrial são introduzidas a cada ano. A maioria é fornecida por empresas que afirmam que aquela ferramenta irá melhorar o gerenciamento de projeto, ou a análise de requisitos, ou a modelagem do projeto, ou geração de código, ou o teste, ou gerenciamento de mudanças, ou qualquer uma das muitas atividades de engenharia de software, ações e tarefas discutidas neste livro. Outras foram desenvolvidas como ofertas de código aberto. A maioria das ferramentas de código aberto concentra-se nas atividades de “programação” com ênfase específica na construção (particularmente a geração de código). Há ainda outras que resultam de esforços de pesquisas em laboratórios de universidades ou do governo. Embora tenham um atrativo em aplicações bastante limitadas, grande parte não está pronta para a aplicação mais ampla da indústria.

Em nível industrial, os pacotes mais abrangentes formam os *ambientes de engenharia de software* (*software engineering environments* — SEE)¹³ que integram uma coleção de ferramentas individuais ao redor de uma base de dados central (repositório). Quando considerado um todo, um SEE integra informações por meio do processo de software e auxilia na colaboração necessária para muitos sistemas grandes e complexos baseados em software. Mas os ambientes atuais não são facilmente extensíveis (é difícil integrar uma ferramenta COTS que não faz parte do pacote) e tendem a ser de uso geral (não são específicos de domínio de aplicação). Há também um retardo de tempo substancial entre a introdução de novas soluções de tecnologia (por exemplo, desenvolvimento de software controlado por modelo) e a disponibilidade de SEEs viáveis que suportam a nova tecnologia.

As tendências futuras em ferramentas de software seguirão dois caminhos distintos — um *caminho focalizado no homem* que responde a algumas das “tendências leves” discutidas na Seção 31.3, e o caminho centrado na tecnologia que trata de novas tecnologias (Seção 31.4) à medida que são introduzidas e adotadas. Nas próximas seções, examinaremos rapidamente cada um desses caminhos.

¹³ É usado também o termo *ambiente de desenvolvimento integrado* (*integrated development environment* — IDE).

31.5.1 Ferramentas que respondem a tendências leves

As tendências leves discutidas na Seção 31.3 — a necessidade de gerenciar a complexidade, acomodar requisitos emergentes, estabelecer modelos de processo que aceitam mudanças, coordenar equipes globais com um *mix* de talentos que varia, entre outras coisas — sugere uma nova era em que suporte de ferramentas para colaboração de interessados se tornará tão importante quanto o suporte de ferramentas para tecnologia. Mas que tipo de conjunto de ferramentas suporta essas tendências?

Um exemplo de pesquisa nessa área é o GENESIS — um ambiente generalizado, de código aberto, criado para suportar trabalho de engenharia de software colaborativo [Ave04]. O ambiente GENESIS pode ou não ter um uso difundido, mas seus elementos básicos são representativos da direção dos SEEs colaborativos que irão evoluir para suportar as tendências leves descritas neste capítulo.

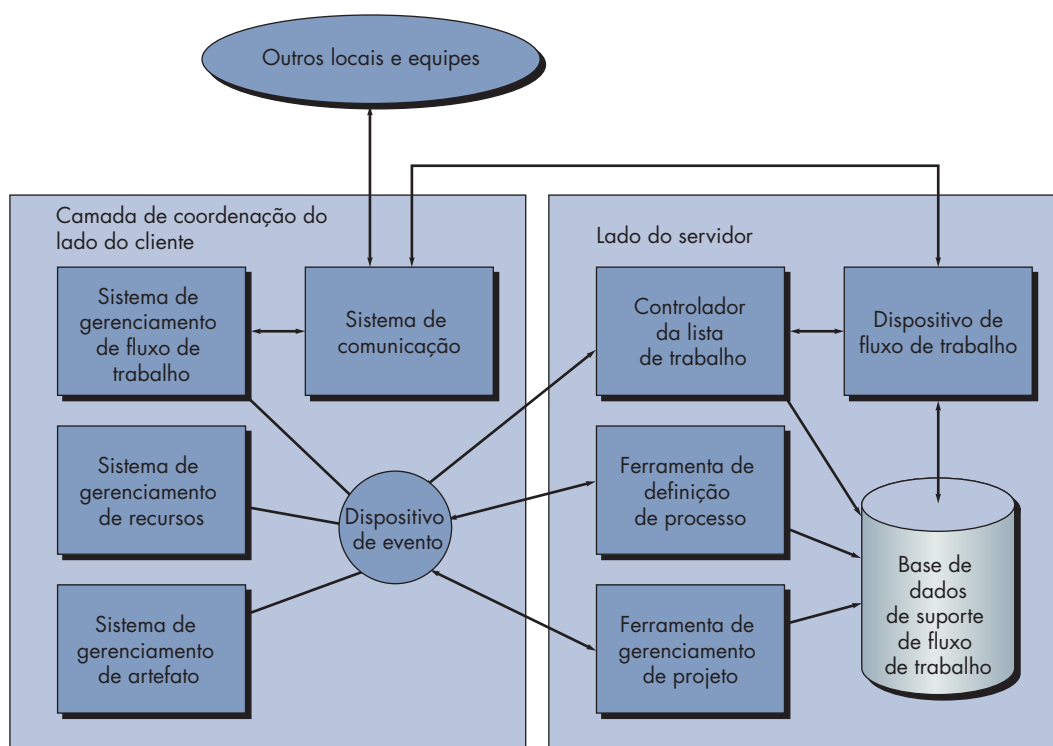
Um SEE colaborativo “suporta cooperação e comunicação entre engenheiros de software pertencentes a equipes de desenvolvimento distribuído envolvidas em modelar, controlar e medir processos de desenvolvimento de software e de manutenção. E ainda mais, inclui uma função de gerenciamento de artefato que armazena e controla artefatos de software (produtos acabados) produzidos por diferentes equipes no decorrer de seu trabalho” [Bol02].

A Figura 31.4 ilustra uma arquitetura para um SEE colaborativo. A arquitetura, baseada no ambiente GENESIS [Ave04], é construída com subsistemas integrados dentro de um cliente Web comum e é complementada por componentes baseados em servidor que proporcionam suporte para todos os clientes. Cada organização de desenvolvimento tem seus próprios subsistemas no lado do cliente que se comunica com outros clientes. De acordo com a Figura 31.4, um *sub-sistema de gerenciamento de recursos* (*resource management subsystem*) controla a alocação de recursos humanos para diferentes projetos ou subprojetos; um *sistema de gerenciamento de artefatos* (*work product management system*) é “responsável pela criação, modificação, exclusão”,

FIGURA 31.4

Arquitetura SEE colaborativa

Fonte: Adaptado de (Ave04)



indexação, pesquisa e armazenamento de todos os artefatos de engenharia de software [Ave04]; um *subsistema de gerenciamento de fluxo de trabalho* (*workflow management subsystem*) coordena a definição, a ocorrência (*instantiation*) e implementação das atividades, ações e tarefas de processo de software; um dispositivo de evento “coleta eventos” que ocorrem durante o processo de software (por exemplo, uma revisão bem-sucedida de um artefato, o término de um teste de unidade de um componente) e notifica os outros; um *sistema de comunicações* (*communication system*) suporta comunicação síncrona e assíncrona entre as equipes distribuídas.

No lado do servidor, quatro componentes compartilham uma base de dados de suporte de fluxo de trabalho. Os componentes implementam as seguintes funções:

- *Definição de processo* – ferramenta que permite à equipe definir novas atividades de processo, ações ou tarefas e define as regras que governam como esses elementos de processo interagem uns com os outros e os produtos acabados que eles geram.
- *Gerenciamento de projeto* – conjunto de ferramentas que permite à equipe criar um plano de projeto e coordenar o plano com outras equipes ou projetos.
- *Dispositivo de fluxo de trabalho* (*workflow engine*) – “interage com o dispositivo de evento para propagar eventos que são relevantes para a execução de processos cooperativos executados em outros locais” [Ave04].
- *Manipulador de lista de trabalhos* – interage com a base de dados do lado do servidor para fornecer ao engenheiro de software informações sobre a tarefa que está em execução no momento ou qualquer tarefa futura que seja derivada do trabalho que esteja em execução no momento.

Embora a arquitetura de um SEE colaborativo possa variar consideravelmente em relação àquele discutido nesta seção, os elementos funcionais básicos (sistemas de gerenciamento e componentes) parecerão atingir o nível de coordenação necessário para um projeto distribuído de engenharia de software.

31.5.2 Ferramentas que lidam com as tendências tecnológicas

A agilidade na engenharia de software (Capítulo 3) é obtida quando os interessados trabalham em equipe. Portanto, a tendência para os SEEs colaborativos (Seção 31.5.1) produzirá benefícios mesmo quando o software é desenvolvido localmente. Mas qual das ferramentas de tecnologia complementa o sistema e componentes que fortalecem uma melhor colaboração?

Uma das tendências dominantes é a criação de um conjunto de ferramentas que suporta desenvolvimento controlado por modelo (Seção 31.4.4) com ênfase no projeto controlado por arquitetura. Oren Novotny [Nov04] sugere que o modelo e não o código-fonte se torne o foco da engenharia de software:

Modelos independentes de plataforma são criados em UML e então passam por vários níveis de transformação para eventualmente se transformarem em código-fonte para uma plataforma específica. É razoável concluir que o modelo, não o arquivo, deverá se tornar a nova unidade de saída. Um modelo tem muitas visões diferentes em diferentes níveis de abstração. No nível mais alto, componentes independentes de plataforma podem ser especificados em análise; no nível mais baixo há uma implementação específica de plataforma que se reduz a um conjunto de classes no código.

Novotny afirma que uma nova geração de ferramentas funcionará em conjunto com um repositório para criar modelos em todos os níveis necessários de abstração, estabelecer relações entre os vários modelos, transformar modelos de um nível de abstração para outro (por exemplo, transformar um modelo de projeto em código-fonte), gerenciar alterações e versões e coordenar ações de controle e garantia de qualidade nos modelos de software.

Além dos ambientes completos de engenharia de software, ferramentas de solução pontual que resolvem tudo, desde reunião de requisitos até refatoração de projeto/código e teste, con-

tinuarão a evoluir e se tornarão mais capazes em funcionalidade. Em algumas situações, ferramentas de modelagem e teste voltadas para um domínio de aplicação específico proporcionarão um melhor benefício quando comparadas com seus equivalentes genéricos.

31.6 RESUMO

As tendências que têm um efeito sobre a tecnologia de engenharia de software muitas vezes originam-se de cenários de negócios, organizacionais, mercado e cultural. Essas “tendências leves” podem influir na direção da pesquisa e da tecnologia derivada em consequência da pesquisa.

Quando uma nova tecnologia é introduzida, ela passa por um ciclo de vida que nem sempre leva a uma aceitação ampla, apesar de as expectativas originais serem elevadas. O grau segundo o qual qualquer tecnologia de engenharia de software ganha uma aceitação ampla está ligado a sua habilidade para resolver os problemas apresentados pelas tendências tanto leves (*soft*) quanto pesadas (*hard*).

Tendências leves — a necessidade cada vez maior de conectividade e colaboração, projetos globais, transferência de conhecimento, o impacto das economias emergentes e a influência da própria cultura humana levam a uma série de desafios que abrangem desde o gerenciamento de complexidade e requisitos emergentes até a manipulação de um *mix* de talentos sempre em mudanças entre equipes de software geograficamente dispersas.

Tendências pesadas — o ritmo sempre acelerado da mudança da tecnologia — surgem do âmbito das tendências leves e afetam a estrutura do software e o escopo dos processos e a maneira pela qual uma estrutura de processo é caracterizada. Desenvolvimento colaborativo, novas formas de engenharia de requisitos, desenvolvimento baseado em modelo e controlado por teste, e projeto pós-moderno mudarão o cenário de métodos. Os ambientes de ferramentas responderão a uma necessidade cada vez maior de comunicação e colaboração e ao mesmo tempo integram soluções pontuais específicas de domínio que podem mudar a natureza atual das tarefas de engenharia de software.

PROBLEMAS E PONTOS A PONDERAR

31.1. Obtenha uma cópia do *best-seller* *The tipping point*, de Malcolm Gladwell (disponível via Google Book Search) e discuta como suas teorias se aplicam à adoção das novas tecnologias de engenharia de software.

31.2. Por que o software aberto apresenta um desafio às abordagens convencionais de engenharia de software?

31.3. Reveja o *ciclo da excelência para tecnologias emergentes* do Gartner Group. Selecione um produto de tecnologia bem conhecido e apresente um breve histórico que ilustre como ele se comportou ao longo da curva. Selecione outro produto de tecnologia bem conhecida que não seguiu o caminho sugerido pela curva da excelência.

31.4. O que é uma “tendência leve”?

31.5. Você enfrenta um problema extremamente complexo que vai requerer uma solução demorada. Como trata a complexidade desse problema e como propõe uma solução?

31.6. O que são “requisitos emergentes” e por que eles representam um desafio para os engenheiros de software?

31.7. Selecione um trabalho de desenvolvimento de código aberto (que não o Linux) e apresente um breve histórico de sua evolução e sucesso relativo.

31.8. Descreva como o processo de software mudará durante a próxima década.

31.9. Você trabalha em Los Angeles e participa de uma equipe global de engenharia de software. Você e colegas em Londres, Mumbai, Hong Kong e Sydney devem editar uma especifi-

cação de requisitos de 245 páginas para um grande sistema. A primeira edição deve ser feita em três dias. Descreva o conjunto ideal de ferramentas on-line que lhe possibilitaria colaborar eficazmente.

31.10. Descreva o desenvolvimento de software controlado por modelo. Faça o mesmo para o desenvolvimento controlado por teste.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Livros que discutem o caminho para software e computação abrangem uma grande variedade de assuntos técnicos, científicos, econômicos, políticos e sociais. Kurweil (*The Singularity Is Near*, Penguin Books, 2005) apresenta uma visão atraente de um mundo que mudará de formas muito profundas nos meados deste século. Sterling (*Tomorrow Now*, Random House, 2002) lembra-nos de que o progresso real raramente é ordenado e eficiente. Teich (*Technology and the Future*, Wadworth, 2002) apresenta ensaios sobre o impacto social da tecnologia e como a cultura em mudança dá forma à tecnologia. Naisbitt, Philips e Naisbitt (*High Tech/High Touch*, Nicholas Brealey, 2001) observam que muitos de nós se tornaram “intoxicados” com alta tecnologia e que a “grande ironia da era de alta tecnologia é que nos tornamos escravos de dispositivos que se destinavam a nos dar a liberdade”. Zey (*The Future Factor*, McGraw-Hill, 2000) discute cinco forças que definirão o destino humano durante este século. A obra de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) foi um *best-seller* nos meados da década de 1990 e continua a fornecer uma visão esclarecedora da computação e seu impacto geral.

À medida que o software se torna parte do cenário de quase todas as facetas de nossas vidas, a “cyber ética” evoluiu como um importante tópico de discussão. Livros de Spinello (*Cyberethics: Morality and Law in Cyberspace*, Jones & Bartlett Publishers, 2002), Halbert e Ingulli (*Cyberethics*, South-Western College Publishers, 2001) e Baird e seus colegas (*Cyberethics: Social and Moral Issues in the Computer Age*, Prometheus Books, 2000) consideram o tópico em detalhe. O governo dos Estados Unidos publicou um volumoso relatório em CD-ROM (*21st Century Guide to Cybercrime*, Progressive Management, 2003) que considera todos os aspectos do crime no computador, problemas de propriedade intelectual e o National Infrastructure Protection Center (NIPC).

Uma grande variedade de fontes de informação sobre os rumos futuros das tecnologias relacionadas ao software e engenharia de software está disponível na Internet. Uma lista atualizada das referências da Web relevantes sobre as futuras tendências na engenharia de software pode ser encontrada no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

**CONCEITOS-
-CHAVE**

conhecimento	723
ética.	725
espectro de	
informações	723
futuro.	721
profissionais	722
software revisitado .	722

Nos 31 capítulos deste livro, exploramos um processo para engenharia de software que abrange tanto procedimentos de gestão quanto métodos técnicos, conceitos básicos e princípios, técnicas especializadas, atividades e tarefas orientadas a pessoas, que são próprias à automação, anotações com lápis e papel e ferramentas de software. Argumentamos que medições, disciplina e um foco na agilidade e qualidade resultarão em software que atenda às necessidades do cliente, software que é confiável, software que é manutenível, um software melhor. Contudo, nunca foi dito que a engenharia de software é um remédio para todas as dificuldades.

Enquanto caminhamos para a segunda década deste século, as tecnologias de software e sistemas permanecem um desafio para todos os profissionais de software e todas as empresas que criam sistemas baseados em computadores. Embora tenhamos escrito essas palavras com a visão do século 20, Max Hopper [Hop90] descreve com precisão o estado atual das coisas:

Uma vez que as mudanças na tecnologia de informação estão se tornando tão rápidas e intormentadas e as consequências de ficar para trás são tão irreversíveis, ou as empresas dominam a tecnologia ou morrem... Pense nisso como uma escravidão tecnológica (technology treadmill). As empresas terão de lutar cada vez mais para manter sua posição.

As mudanças na tecnologia de engenharia de software são sem dúvida “rápidas e inclementes”, ao mesmo tempo o progresso é com frequência bastante lento. Até que seja tomada a decisão de adotar um novo processo, método ou ferramenta, seja realizado o treinamento necessário para entender sua aplicação e seja introduzida a tecnologia na cultura de desenvolvimento de software, algo mais novo (e até melhor) surge, e o processo todo começa novamente.

Uma coisa que aprendemos durante muitos anos neste campo é que os profissionais de engenharia de software são “conscientes de sua situação”. O caminho à frente estará entulhado de carcaças de novas e excitantes tecnologias (a última moda) que nunca tiveram sucesso (apesar da quantidade de “novidades”). O futuro será formado por tecnologias mais modestas que de certa forma modificam a direção e a amplitude do caminho. Abordei algumas dessas tecnologias no Capítulo 31.

Neste capítulo conclusivo vamos adotar uma visão mais ampla e considerar onde estamos e para onde estamos indo a partir de uma perspectiva mais filosófica.

PANORAMA

O que é? Ao chegarmos ao fim de uma jornada relativamente longa pela engenharia de software, é hora de colocarmos os fatos em perspectiva e fazer alguns comentários finais.

Quem realiza? Autores como eu. Quando você chega ao fim de um livro longo e desafiador, é ótimo poder reunir os fatos de maneira clara.

Por que é importante? É sempre bom lembrar onde estivemos e pensar para onde vamos.

Quais são as etapas envolvidas? Farei considerações sobre o que vimos e vou tratar de alguns dos

assuntos importantes e algumas direções para o futuro.

Qual é o artefato? Uma discussão que o ajudará a entender o contexto real de desenvolvimento de software.

Como garantir que o trabalho foi realizado corretamente? Isso é difícil de conseguir em tempo real. Somente daqui a alguns anos é que se pode dizer se conceitos, princípios, métodos e técnicas de engenharia de software discutidos neste livro o ajudaram a tornar-se um engenheiro de software melhor.

32.1 A IMPORTÂNCIA DO SOFTWARE REVISITADA

A importância do software pode ser definida de várias maneiras. No Capítulo 1, o software foi caracterizado como um diferenciador. A função proporcionada pelo software diferencia produtos, sistemas e serviços e traz uma vantagem competitiva no mercado. Mas o software é mais do que um diferenciador. Quando considerados como um todo, os produtos de engenharia de software geram o bem de consumo mais importante que qualquer indivíduo, negócio ou governo pode adquirir – a informação.

No Capítulo 31, abordou-se rapidamente a computação aberta – inteligência ambiente, aplicações ligadas a contexto e computação invasiva/onipresente –, uma direção que mudará fundamentalmente nossa percepção de computadores, as coisas que fazemos com eles (e que eles fazem para nós) e nossa percepção das informações como um guia, um bem e uma necessidade. Destaquei também que o software necessário para suportar a computação aberta apresentará novos desafios para os engenheiros de software. Porém mais importante ainda, a futura invasão do software de computador apresentará desafios ainda mais significativos para a sociedade como um todo. Sempre que uma tecnologia tem um amplo impacto – que pode salvar vidas ou colocá-las em perigo, criar negócios ou destruí-los, informar os líderes dos governos ou confundi-los –, deve ser “manuseada com cuidado”.

32.2 PROFISSIONAIS E A MANEIRA COMO CONSTROEM SISTEMAS

“Choque do futuro é o ambiente de estresse e desorientação que criamos ao redor dos indivíduos, submetendo-os a inúmeras mudanças em um período de tempo muito curto.”

Alvin Toffler

O software necessário para os sistemas de alta tecnologia torna-se cada vez mais complexo a cada ano que passa, e o tamanho dos programas resultantes aumenta proporcionalmente. O rápido crescimento do tamanho dos programas “médios” não nos causaria problemas se não fosse por um simples fato: à medida que aumenta o tamanho do programa, o número de pessoas envolvidas com o trabalho também aumenta.

A experiência mostra que quando o número de profissionais em uma equipe de projeto de software aumenta, a produtividade geral do grupo pode ficar prejudicada. Um meio de contornar o problema está na criação de uma série de equipes de engenharia de software, dividindo assim as pessoas em grupos individuais de trabalho. No entanto, enquanto o número de equipes de engenharia de software cresce, a comunicação entre elas se torna tão difícil e morosa quanto a estabelecida entre os indivíduos. Pior ainda, a comunicação (entre indivíduos ou equipes) tende a ser ineficiente – muito tempo é gasto transferindo-se pouco conteúdo e, com frequência, informações importantes “caem no vazio”.

Se a comunidade de engenharia de software tem de enfrentar efetivamente o dilema das comunicações, o caminho para os engenheiros de software deve incluir mudanças radicais na maneira como os profissionais e as equipes se comunicam uns com os outros. No Capítulo 31, abordei os ambientes colaborativos que podem proporcionar melhoras significativas na maneira como as equipes se comunicam.

Na análise final, a comunicação é a transferência de conhecimento, e a aquisição (e transferência) de conhecimento está mudando profundamente. Conforme os dispositivos de busca se tornam cada vez mais sofisticados e as aplicações para Web 2.0 proporcionam uma melhor sinergia, a maior biblioteca do mundo de artigos de pesquisa e relatórios, tutoriais, comentários e referências torna-se facilmente acessível e utilizável.

Se a história pode servir de referência, podemos afirmar que as pessoas em si não mudarão. No entanto, a maneira pela qual elas se comunicam, o ambiente no qual trabalham, o modo como adquirem conhecimentos, os métodos e ferramentas que utilizam, a disciplina que aplicam e, portanto, a cultura geral para o desenvolvimento futuro mudarão de forma significativa e ainda mais profunda.

32.3 NOVOS MODOS DE REPRESENTAR AS INFORMAÇÕES

“A melhor preparação para fazer um bom trabalho amanhã é fazer um bom trabalho hoje.”

Elbert Hubbard

Na história da computação, houve uma mudança sutil na terminologia usada para descrever o trabalho de desenvolvimento de software para a comunidade dos negócios. Há 40 anos, o termo *processamento de dados* era a frase operativa para descrever o uso dos computadores em um contexto comercial. Hoje, o processamento de dados deu origem a outra expressão – tecnologia da informação –, que implica a mesma coisa, mas apresenta uma mudança sutil no foco. A ênfase não é apenas no processamento de grandes quantidades de dados, e sim na extração de informações importantes desses dados. Obviamente, essa sempre foi a intenção, mas a transformação na tecnologia reflete uma alteração muito mais importante na filosofia de gerenciamento.

Quando as aplicações de software são discutidas hoje, as palavras *dados*, *informações* e *conteúdo* ocorrem repetidamente. Encontramos a palavra *conhecimento* em algumas aplicações de inteligência artificial, mas seu uso é relativamente raro. Praticamente ninguém discute sabedoria no contexto de aplicações de software.

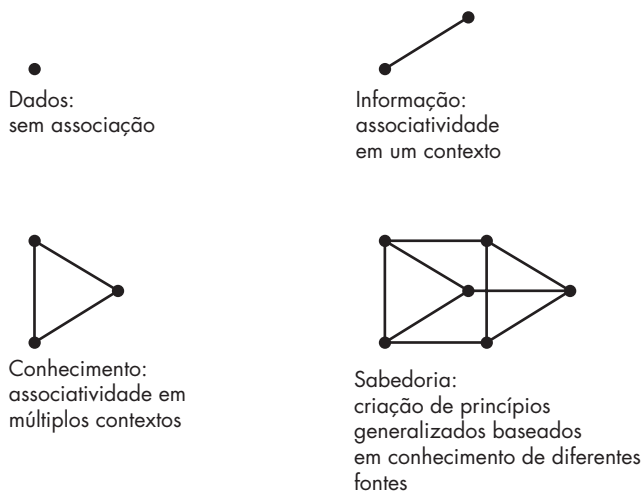
Dados são informações em estado bruto – coleções de fatos que devem ser processadas para ter um significado. As informações surgem associando-se fatos em determinado contexto. Conhecimento associa informações obtidas em um contexto com informações obtidas em outro. Por fim, a sabedoria ocorre quando princípios generalizados são derivados de conhecimentos dispersos. Cada uma dessas quatro visões da “informação” é representada esquematicamente na Figura 32.1.

Até hoje, grande parte do software tem sido criada para processar dados ou informações. Engenheiros de software estão agora igualmente preocupados com sistemas que processam conhecimento.¹ Conhecimento é bidimensional. Informações coletadas sobre uma variedade de tópicos relacionados e não relacionados são conectadas para formar um corpo do fato que chamamos de *conhecimento*. A chave é nossa habilidade em associar informações de uma variedade de origens que podem não ter qualquer conexão óbvia e combiná-las de maneira que nos proporcione um benefício distinto.²

Para ilustrar a progressão dos dados para o conhecimento, considere os dados do senso indicando que o número de nascimentos em 1996 nos Estados Unidos foi de 4,9 milhões. Esse número representa o valor de um dado. Relacionando esse dado com os números de nascimentos nos 40 anos anteriores, podemos gerar uma informação útil – os *baby boomers* dos anos 1950 e início

FIGURA 32.1

Um espectro de “informações”



¹ O rápido crescimento das tecnologias de obtenção de dados e tratamento de dados reflete essa tendência em crescimento.

² A semântica da Web (Web 2.0) permite a criação de “mashups” que podem proporcionar um mecanismo fácil para conseguir isso.

“Sabedoria é o poder que nos permite usar o conhecimento para nosso próprio benefício e para o benefício dos outros.”

Thomas J. Watson

dos anos 1960 que agora estão envelhecendo fizeram um último esforço para ter seus filhos antes do fim da idade fértil. Além disso, os indivíduos da geração X (*gen-Xers*) entraram em sua idade fértil. Os dados do senso podem então ser ligados a outras partes de informações aparentemente não relacionadas. Por exemplo, o número atual de professores da escola primária que se aposentarão durante a próxima década, o número de estudantes que estarão se formando no primário e secundário, a pressão sobre os políticos para baixar os impostos e, portanto, limitar os aumentos de salários aos professores. Todas essas informações podem ser combinadas para formular uma representação do conhecimento – haverá uma pressão significativa sobre o sistema de educação nos Estados Unidos no início do século 21 e essa pressão continuará por muitas décadas. Usando esse conhecimento, pode surgir uma oportunidade de negócios. Pode haver uma oportunidade significativa de desenvolver novos modos de aprendizado mais eficazes e menos dispendiosos do que a forma atual.

O futuro do software leva a sistemas que processam conhecimento. Estivemos processando dados usando computadores por mais de 50 anos e extraindo informações por mais de três décadas. Um dos desafios mais importantes para a comunidade de engenharia de software é criar sistemas que possibilitem o próximo passo ao longo do espectro – sistemas que extraem conhecimento dos dados e informações de maneira prática e benéfica.

32.4 A VISÃO NO LONGO PRAZO

Na Seção 32.3, sugeri que o futuro leva a sistemas que “processam conhecimento”. Mas o futuro da computação em geral e dos sistemas baseados em software em particular podem levar a eventos consideravelmente mais profundos.

Em um fascinante livro, leitura obrigatória para todo profissional envolvido em tecnologias da computação, Ray Kurzweil [Kur05] afirma que já chegamos a uma época em que “o ritmo das mudanças tecnológicas será tão rápido, seu impacto tão profundo, que a vida humana será transformada irreversivelmente”. Kurzweil³ apresenta um argumento convincente de que estamos atualmente no “joelho” de uma curva de crescimento exponencial que nos levará a enormes avanços na capacidade de computação durante as próximas duas décadas. Se combinados com avanços equivalentes em nanotecnologia, genética e robótica, pode-se chegar a uma época nos meados deste século em que a distinção entre humanos (como os conhecemos hoje) e máquinas começa a se confundir – época na qual a evolução humana se acelera de modo assustador (para alguns) e espetacular (para outros).

Aproximadamente em 2030, Kurzweil afirma que a capacidade de computação e os requisitos de software serão suficientes para modelar todo aspecto do cérebro humano – todas as conexões físicas, processos analógicos e camadas químicas. Quando isso ocorrer, os seres humanos terão atingido um “poderoso AI (inteligência artificial)”, e, em consequência disso, máquinas que realmente pensam (usando os termos convencionais de hoje). Mas haverá uma diferença fundamental. Os processos do cérebro humano são excessivamente complexos e fracamente ligados às fontes externas de informação. Eles também são lentos em computação, mesmo em comparação com a tecnologia atual. Se ocorrer a emulação completa do cérebro humano, o “pensamento” ocorrerá a velocidades milhares de vezes maiores do que seu correspondente humano com conexões íntimas a um mar de informações (pense na Web de hoje como um exemplo primitivo). O resultado é... Bem, é tão fantástico que é melhor deixar que o próprio Kurzweil descreva.

É importante notar que nem todos acreditam que o futuro que Kurzweil descreve é uma coisa boa. Em um ensaio agora famoso intitulado “O futuro não precisa de nós” (*The Future Doesn't Need Us*), Bill Joy [Joy00], um dos fundadores da Sun Microsystems, diz que a “robótica, a enge-

3 É importante notar que Kurzweil não é um escritor de ficção científica comum ou um futurista sem portfólio. Ele é um tecnólogo sério que (segundo a Wikipedia) “tem sido um pioneiro nos campos do reconhecimento óptico de caractere (OCR), síntese texto-para-fala, tecnologia de reconhecimento de voz e instrumentos de teclado eletrônico”.

nharia genética e a nanotecnologia estão fazendo dos humanos uma espécie ameaçada”. Seus argumentos prevendo uma antiutopia tecnológica representam um contra-argumento ao futuro utópico previsto por Kurzweil. Ambos devem ser considerados seriamente quando os engenheiros de software assumem um dos papéis importantes definindo um futuro para a raça humana.

32.5 A RESPONSABILIDADE DO ENGENHEIRO DE SOFTWARE

A engenharia de software evoluiu tornando-se uma profissão mundialmente respeitada. Como profissionais, engenheiros de software devem se orientar por um código de ética que rege o trabalho que fazem e os produtos que criam. Uma força-tarefa da ACM/IEEE-CS (ACM/IEEE-CS Joint Task Force) produziu um código de ética e práticas profissionais da engenharia de software (*Software Engineering Code of Ethics and Professional Practices*) (Versão 5.1). O código declara[ACM98]:

Engenheiros de software deverão se comprometer a fazer da análise, especificação, projeto, desenvolvimento, teste e manutenção de software uma profissão benéfica e respeitada. De acordo com seu compromisso com a saúde, segurança e bem-estar do público, engenheiros de software deverão adotar os oito princípios a seguir:

WebRef

Uma discussão completa sobre código de ética ACM/IEEE pode ser encontrada em seeri.etsu.edu/Codes/default.shtm.

1. PÚBLICO – Engenheiros de software deverão agir consistentemente com o interesse público.
2. CLIENTE E EMPREGADOR – Engenheiros de software deverão agir de acordo com o melhor interesse de seu cliente e empregador consistente com o interesse público.
3. PRODUTO – Engenheiros de software deverão garantir que seus produtos e modificações relacionadas atinjam os mais altos padrões profissionais possíveis.
4. JULGAMENTO – Engenheiros de software deverão manter a integridade e independência de seu julgamento profissional.
5. GESTÃO – Gerentes de engenharia de software e líderes devem aderir e promover uma abordagem ética para a gestão do desenvolvimento e manutenção de software.
6. PROFISSÃO – Engenheiros de software devem prestigiar a integridade e reputação da profissão consistente com o interesse público.
7. COLEGAS – Engenheiros de software deverão ser justos e solidários com seus colegas.
8. INDIVIDUALMENTE – Engenheiros de software devem participar do contínuo aprendizado referente à prática de sua profissão e deverão promover uma abordagem ética à prática da profissão.

Embora cada um desses oito princípios seja igualmente importante, surge um tema mais importante: um engenheiro de software deve trabalhar no interesse público. Em nível pessoal, um engenheiro de software deve se guiar pelas seguintes regras:

- Nunca roubar dados para obter vantagens pessoais.
- Nunca distribuir ou vender informações proprietárias obtidas como parte de seu trabalho em um projeto de software.
- Nunca destruir ou modificar maliciosamente os programas, arquivos ou dados de outra pessoa.
- Nunca violar a privacidade de um indivíduo, um grupo ou uma organização.
- Nunca invadir um sistema por esporte ou por lucro.
- Nunca criar ou promover um vírus ou verme de computador.
- Nunca usar a tecnologia de computação para facilitar a discriminação ou assédio.

Durante a última década, certos membros da indústria de software formaram um *lobby* por uma legislação de proteção que [SEE03]: (1) permite às empresas liberar software sem revelar os defeitos conhecidos, (2) isenta os desenvolvedores da responsabilidade por quaisquer danos

resultantes desses defeitos conhecidos, (3) impede que outros descubram defeitos sem permissão do desenvolvedor original, (4) permite a incorporação de software “autoajuda” dentro do produto que pode desabilitar (via comando remoto) a operação do produto, e (5) isenta os desenvolvedores do software com “autoajuda” de danos se o software for desabilitado por um terceiro.

Como toda legislação, o debate em geral concentra-se em aspectos políticos, não tecnológicos. No entanto, muitas pessoas (até mesmo eu) pensam que a legislação protetora, se formulada incorretamente, entra em conflito com o código de ética da engenharia de software, isentando indiretamente os engenheiros de sua responsabilidade de produzir software de alta qualidade.

32.6 COMENTÁRIO FINAL

Já se passaram 30 anos desde que comecei a trabalhar na primeira edição deste livro. Ainda me lembro sentado à minha mesa como jovem professor, escrevendo o manuscrito de um livro sobre um assunto com o qual poucas pessoas se importavam e ainda menos pessoas realmente entendiam. Lembro-me das cartas de rejeição dos editores, que argumentavam (educadamente, mas com firmeza) que nunca haveria um mercado para um livro sobre “engenharia de software”. Felizmente, a McGraw-Hill decidiu tentar,⁴ e o resto, como se costuma dizer, é história.

Durante os últimos 30 anos, este livro mudou significativamente – em escopo, em tamanho, em estilo e em conteúdo. Assim como a engenharia de software, cresceu e (espero) amadureceu durante os últimos anos.

Uma abordagem de engenharia para o desenvolvimento de software de computador é hoje sabedoria convencional. Embora o debate continue sobre o “paradigma certo”, a importância da agilidade, o grau de automação e os métodos mais eficazes, os princípios subjacentes de engenharia de software são agora aceitos em toda a indústria. Por que, então, só recentemente temos visto uma adoção mais ampla?

A resposta, penso, está na dificuldade da transição tecnológica e na mudança cultural que a acompanha. Apesar de muitos de nós apreciarem a necessidade de uma disciplina de engenharia para software, lutamos contra a inércia da prática anterior e nos defrontamos com novos domínios de aplicação (e os desenvolvedores que trabalham nelas) que parecem prontos para repetir os mesmos erros do passado. Para facilitarmos a transição, precisamos de muitas coisas – um processo de software ágil, adaptável e sensível; métodos mais eficazes; ferramentas mais poderosas; melhor aceitação pelos profissionais e apoio dos gerentes; e uma boa dose de educação.

Você pode não concordar com todas as abordagens descritas neste livro. Algumas das técnicas e opiniões são controvertidas; outras devem ser ajustadas para funcionar bem em diferentes ambientes de desenvolvimento de software. No entanto, espero sinceramente que *Engenharia de software: uma abordagem profissional* tenha delineado os problemas que enfrentamos, demonstrado o poder dos conceitos de engenharia de software e tenha proporcionado uma estrutura de métodos e ferramentas.

À medida que avançamos pelo século 21, o software continua a ser o produto e a indústria mais importantes no cenário mundial. Seu impacto e importância têm um amplo alcance. Contudo, uma nova geração de desenvolvedores de software deve enfrentar muitos dos mesmos desafios que gerações anteriores enfrentaram.

Esperemos que as pessoas que enfrentam esse desafio – os engenheiros de software – tenham a sabedoria para desenvolver sistemas que melhorem a condição humana.

⁴ Na realidade, o mérito vai para Peter Freeman e Eric Munson, que convenceram a McGraw-Hill de que valia a pena tentar. Depois de um milhão de cópias, é justo dizer que tomaram uma boa decisão.

**CONCEITOS-
-CHAVE**

dependência	844
diagrama de atividade	853
diagrama de caso de uso	847
diagrama de classe	842
diagrama de comunicação	851
diagrama de distribuição/ instalação	846
diagrama de estado	856
diagrama de sequência	848
stereotype	843
frames de interação	850
generalização	843
Object Constraint Language	859
multiplicidade	844
raias	855

A UML (*Unified Modeling Language – linguagem de modelagem unificada*) é “uma linguagem-padrão para descrever/documentar projeto de software. A UML pode ser usada para visualizar, especificar, construir e documentar os artefatos de um sistema de software-intensivo” [Boo05]. Em outras palavras, assim como os arquitetos criam plantas e projetos para ser usados por uma empresa de construção, os arquitetos de software criam diagramas UML para ajudar os desenvolvedores de software a construir o software. Se você entender o vocabulário da UML (os elementos visuais do diagrama e seus significados), pode facilmente entender e especificar um sistema e explicar o projeto daquele sistema para outros interessados.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desenvolveram a UML na década de 1990 com muita realimentação da comunidade de desenvolvimento de software. A UML combinou um grupo de notações de modelagem concorrentes usadas pela indústria do software na época. Em 1997, a UML 1.0 foi apresentada ao OMG (Object Management Group), uma associação sem fins lucrativos dedicada a manter especificações para ser usadas pela indústria de computadores. A UML 1.0 foi revisada tornando-se a UML 1.1 e adotada mais tarde naquele ano. O padrão atual é a UML 2.0 e agora é um padrão ISO. Em virtude desse padrão ser tão novo, muitas referências mais antigas, como [Gam95] não usam a notação UML.

A UML 2.0 fornece 13 diferentes diagramas para uso na modelagem de software. Neste apêndice, discutiremos apenas os diagramas de *classe*, *distribuição*, *caso de uso*, *sequência*, *comunicação*, *atividade* e *estado*.

Você notará que há muitas características opcionais em diagramas UML. A linguagem UML proporciona essas opções (às vezes obscuras) para que você possa expressar todos os aspectos importantes de um sistema. Ao mesmo tempo, é possível suprimir partes não relevantes ao aspecto que está sendo modelado para evitar congestionar o diagrama com detalhes irrelevantes. Portanto, a omissão de uma característica particular não significa que esteja ausente, mas sim que foi suprimida. Neste apêndice, *não* apresentamos uma discussão exaustiva de todas as características dos diagramas UML. Em vez disso, nos concentramos nas opções-padrão, especialmente as usadas neste livro.

DIAGRAMAS DE CLASSE

Para modelar classes, incluindo seus atributos, operações e relações e associações com outras classes,² a UML tem um *diagrama de classe*. Ele fornece uma visão estática ou estrutural de um sistema, mas não mostra a natureza dinâmica das comunicações entre os objetos das classes no diagrama.

Os elementos principais são caixas, ou seja, ícones usados para representar classes e interfaces. Cada caixa é dividida em partes horizontais. A parte superior contém o nome da classe. A seção do meio lista os atributos da classe. Um *atributo* refere-se a alguma coisa que um objeto daquela classe sabe ou pode fornecer o tempo todo. Atributos são usualmente implementados como campos da classe, mas eles não precisam ser. Poderiam ser valores que a classe calcula a partir de suas variáveis de instância ou valores que a classe pode obter de outros objetos dos quais é composta. Por exemplo, um objeto pode sempre

¹ Este apêndice teve a contribuição de Dale Skrien e foi adaptado de seu livro, *An introduction to object-oriented design and design patterns in Java* (McGraw-Hill, 2008). Todo o conteúdo é usado com permissão.

² Se você não estiver familiarizado com os conceitos orientados a objeto, é apresentada uma breve introdução no Apêndice 2.

saber a hora atual e ser capaz de retorná-la sempre que for solicitado. Portanto, seria apropriado listar a hora atual como um atributo daquela classe de objetos. No entanto, o objeto muito provavelmente não teria a hora armazenada em uma de suas variáveis de instância, porque precisaria continuamente atualizar aquele campo. Em vez disso, o objeto poderia calcular a hora atual (por exemplo, por meio da consulta a objetos de outras classes) no momento em que a hora é requisitada. A terceira seção do diagrama de classes contém as operações ou comportamentos da classe. Uma *operação* refere-se ao que os objetos da classe podem fazer. Usualmente é implementada como um método da classe.

A Figura A1.1 apresenta um simples exemplo de uma classe **Thoroughbred** que modela cavalos puros-sangues. Ela mostra três atributos – **mother**, **father** e **birthyear**. Os diagramas também mostram três operações: *getCurrentAge()*, *getFather()* e *getMother()*. Pode haver outros atributos e operações suprimidos não mostrados no diagrama.

Cada atributo pode ter um nome, um tipo e um nível de visibilidade. O tipo e a visibilidade são opcionais. O tipo vem após o nome e é separado por dois-pontos. A visibilidade é indicada precedendo pelo sinal –, #, ~ ou +, indicando respectivamente, visibilidade *private*, *protected*, *package* ou *public*. Na Figura A1.1, todos os atributos têm visibilidade *private*, conforme indica o sinal de menos (–). Você pode também especificar que um atributo é estático ou de classe usando sublinhado. Cada operação pode também ser mostrada com um nível de visibilidade, parâmetros com nomes e tipos e um tipo de retorno.

Uma classe abstrata ou método abstrato é indicado pelo uso de itálico no nome da classe no diagrama de classes. Como exemplo, veja a classe **Horse** na Figura A1.2. Uma interface é indicada acrescentando-se a frase “«interface»” (chamada de *stereotype*) acima do nome. Veja a interface **OwnedObject** na Figura A1.2. Uma interface também pode ser representada graficamente por um círculo vazio.

FIGURA A1.1

Um diagrama para a classe **Thoroughbred**

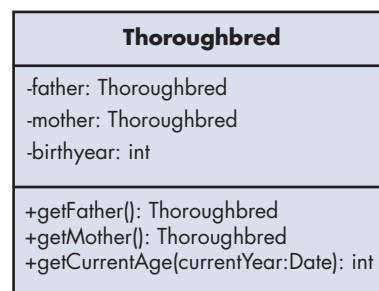
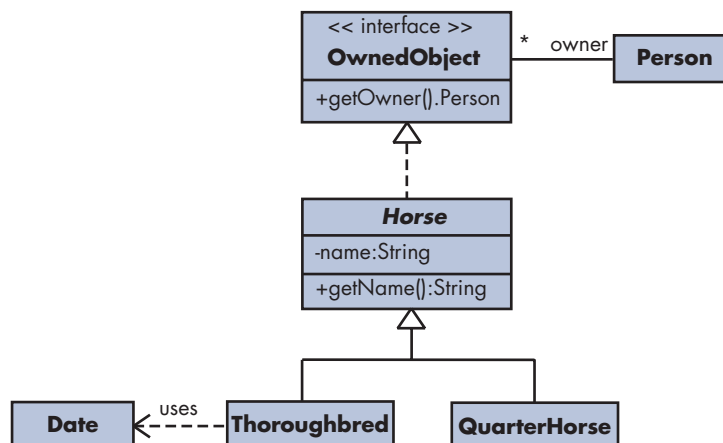


FIGURA A1.2

Um diagrama de classe referente a cavalos



Vale mencionar que o ícone que representa uma classe pode ter outras partes opcionais. Por exemplo, uma quarta seção na parte inferior da caixa de classe pode ser usada para listar as responsabilidades da classe. Essa seção é particularmente útil quando se faz a transição dos cartões CRC (Capítulo 6) para diagramas de classe em que as responsabilidades listadas nos cartões CRC podem ser acrescentadas à quarta seção na caixa da classe no diagrama UML antes de que os atributos e operações que executam essas responsabilidades sejam criados. Essa quarta seção não é mostrada em qualquer uma das figuras neste apêndice.

Os diagramas de classe também podem exibir relações entre classes. Uma classe que seja subclasse de outra classe é conectada a ela por uma seta com uma linha sólida como eixo e com uma ponta triangular vazia. A seta aponta da subclasse para a superclasse. Em UML, uma relação como essa é chamada de *generalização*. Por exemplo, na Figura A1.2, as classes **Thoroughbred** e **QuarterHorse** são exibidas como subclasses da classe abstrata **Horse**. Uma seta tendo como eixo uma linha tracejada indica implementação de interface. Em UML, esse tipo de relação é chamada de *realização*. Na Figura A1.2, a classe **Horse** implementa ou realiza a interface **OwnedObject**.

A *associação* entre duas classes indica que há uma relação estrutural entre elas. Associações são representadas por linhas sólidas. Uma associação tem muitas partes opcionais. Ela pode ser rotulada, assim como cada uma de suas extremidades, para indicar o papel de cada classe na associação. Por exemplo, na Figura A1.2, há uma associação entre **OwnedObject** e **Person** na qual **Person** desempenha o papel de proprietário (*owner*). Setas em qualquer uma ou ambas as extremidades de uma linha de associação indicam navegabilidade. Além disso, cada extremidade da linha de associação pode ter um valor de multiplicidade. Navegabilidade e multiplicidade são explicadas em maior detalhe mais à frente nesta seção. Uma associação pode também conectar uma classe com ela própria, usando um laço. Desse modo, uma associação indica a conexão de um objeto da classe com outros objetos da mesma classe.

A associação com uma seta em uma extremidade indica navegabilidade unidirecional. A seta significa que de uma classe pode-se facilmente acessar a segunda classe associada para a qual aponta a associação. A partir da segunda classe, não se pode necessariamente acessar com facilidade a primeira classe. Outra maneira de pensar sobre isso é que a primeira classe tem conhecimento da segunda classe, enquanto o objeto da segunda classe não necessariamente conhece a primeira classe. Uma associação sem setas em geral indica uma associação bidirecional, que é o que se pretendia na Figura A1.2, mas podia também significar apenas que a navegabilidade não é importante e foi deixada de lado.

Deve-se notar que um atributo de uma classe é muito parecido com uma associação da classe com o tipo de classe do atributo. Isto é, para indicar que uma classe tem uma propriedade chamada “nome” do tipo `String`, poderíamos mostrar aquela propriedade como um atributo, como na classe **Horse** na Figura A1.2. Como alternativa, poderíamos criar uma associação unidirecional da classe **Horse** para a classe **String**, sendo “nome” o papel da classe **String**. A abordagem de atributo é melhor para tipos de dados primitivos, enquanto a abordagem associação muitas vezes é melhor se a classe da propriedade desempenha um papel importante no projeto, caso em que é muito bom ter uma caixa de classe para aquele tipo.

Uma relação de *dependência* representa outra conexão entre classes e é indicada por uma linha tracejada (com setas opcionais nas extremidades e com rótulos opcionais). Uma classe depende de outra se alterações na segunda classe podem requerer alterações na primeira classe. Uma associação de uma classe para outra automaticamente indica uma dependência. Não é necessária uma linha pontilhada entre classes se já houver uma associação entre elas. No entanto, para uma relação transiente (uma classe que não mantém relação de longo prazo com outra classe, mas usa aquela classe ocasionalmente), podemos colocar uma linha tracejada da primeira classe para a segunda. Por exemplo, na Figura A1.2, a classe **Thoroughbred** usa a classe **Date** sempre que é invocado o método `getCurrentAge()`, e assim a dependência é chamada de “uses”.

A *multiplicidade* de extremidade de uma associação indica o número de objetos daquela classe associados a outra classe. Uma multiplicidade é especificada por um inteiro não negativo ou

por um intervalo de inteiros. Uma multiplicidade especificada como “0..1” significa que há 0 ou 1 objeto na extremidade da associação. Por exemplo, cada pessoa no mundo tem um número de Seguro Social ou não tem tal número (especialmente se não forem cidadãos americanos), e assim uma multiplicidade de 0..1 poderia ser usada em uma associação entre uma classe **Person** e uma classe **SocialSecurityNumber** no diagrama de classes. A multiplicidade especificada por “1..*” significa um ou mais, e a multiplicidade especificada por “0..*” ou apenas “*” significa zero ou mais. Usou-se um * como multiplicidade na extremidade **OwnedObject** da associação com a classe **Person** na Figura A1.2 porque uma **Person** pode possuir zero ou mais objetos.

Se a extremidade de uma associação apresenta multiplicidade maior do que 1, os objetos da classe aos quais se faz referência na extremidade da associação estão provavelmente armazenados em uma coleção, como um conjunto ou uma lista ordenada. Poderíamos também incluir a própria classe de coleção no diagrama UML, mas uma classe desse tipo usualmente é desconsiderada e assume-se, implicitamente, que esteja lá devido à multiplicidade da associação.

Uma *agregação* é um tipo especial de associação representada por um losango vazio em uma extremidade do ícone. Ela indica uma relação “todo/parte”, em que a classe para a qual a seta aponta é considerada uma “parte” da classe na extremidade do losango da associação. Uma *composição* é uma agregação indicando forte relação de propriedade entre as partes. Em uma composição, as partes vivem e morrem com o proprietário porque não têm um papel a desempenhar no sistema de software independente do proprietário. Veja na Figura A1.3 exemplos de agregação e composição.

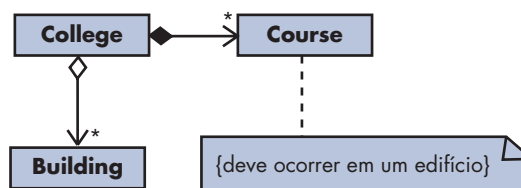
Uma classe **College** tem uma agregação de objetos **Building**, que representam os edifícios que formam o *campus*. A classe **College** tem também uma coleção de cursos. Mesmo que a universidade fechasse, os edifícios ainda continuariam a existir (supondo que a universidade não fosse fisicamente destruída) e poderiam ser usados para outros propósitos, mas um objeto **Course** não tem utilidade fora da universidade na qual está sendo oferecido. Se a universidade (**College**) deixasse de existir como uma entidade de negócios, o objeto **Course** não teria mais utilidade e, portanto, também deixaria de existir.

Outro elemento comum em diagramas de classes é uma anotação (*note*) representada por um caixa com um canto dobrado e é conectada a outros ícones por uma linha pontilhada. Ela pode ter conteúdo arbitrário (texto e gráficos) e é similar aos comentários nas linguagens de programação. Pode ter comentários sobre o papel de uma classe ou restrições que todos os objetos daquela classe devem satisfazer. Se o conteúdo for uma restrição, estará entre chaves. Observe a restrição anexada à classe **Course** na Figura A1.3.

DIAGRAMAS DE DISPONIBILIZAÇÃO

Diagramas de distribuição focalizam a estrutura de um sistema de software e são úteis para mostrar a distribuição física de um sistema de software entre plataformas de hardware e ambientes de execução. Por exemplo, suponha que você esteja desenvolvendo um pacote de renderização gráfica baseado na Web. Os usuários do seu pacote de software usarão o navegador Web para acessar o seu site e introduzir as informações de renderização. O seu site irá renderizar uma imagem gráfica de acordo com as especificações do usuário e a enviará de volta ao usuário. Como a renderização gráfica pode ser cara em termos de computação, você decide mudar a renderização para fora do servidor Web, colocando-a em uma plataforma separada. Portanto,

FIGURA A1.3
As relações entre
Colleges, Courses
e Buildings



haverá três dispositivos de hardware envolvidos no seu sistema: o cliente Web (o computador do usuário executando um navegador), o computador que está hospedando o servidor Web e o computador que está hospedando o dispositivo de renderização.

A Figura A1.4 mostra o diagrama de distribuição para um pacote de software como esse. Neles, os componentes de hardware são desenhados como caixas com o título “<<device>>”. Os caminhos de comunicação entre os componentes de hardware são traçados com linhas com títulos opcionais. Na Figura A1.4, os caminhos são identificados com o protocolo de comunicação e o tipo de rede usada para conectar os dispositivos.

Cada nó em um diagrama de distribuição pode também ser anotado com detalhes sobre o dispositivo. Por exemplo, na Figura A1.4, é desenhado o navegador cliente para mostrar que contém um artefato, que é o software do navegador Web. Artefato é tipicamente um arquivo contendo software executando em um dispositivo. Você pode também especificar valores rotulados, como mostra a Figura A1.4 no nó do servidor Web. Esses valores definem o fornecedor do servidor Web e o sistema operacional usado pelo servidor.

Diagramas de distribuição podem também mostrar os nós do ambiente de execução, desenhados em caixas contendo o rótulo “<<ambiente de execução>>”. Esses nós representam sistemas, como os sistemas operacionais, que podem hospedar outro software.

DIAGRAMAS DE CASOS DE USO

Casos de uso (Capítulos 5 e 6) e o *diagrama de caso de uso* ajudam a determinar a funcionalidade e as características do software sob o ponto de vista do usuário. Para uma ideia de como os casos de uso e os diagramas de caso de uso funcionam, vamos criar alguns deles para um aplicativo de software para gerenciar arquivos de música digital, similar ao software iTunes da Apple.

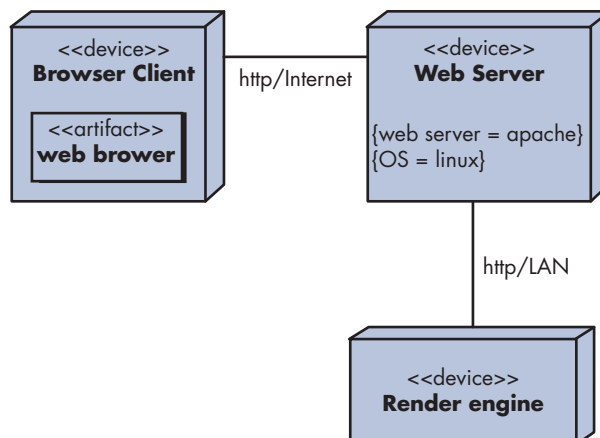
Algumas das coisas que o software pode fazer são:

- Baixar um arquivo de música MP3 e armazená-lo na biblioteca do aplicativo.
- Capturar a música e armazená-la na biblioteca do aplicativo.
- Gerenciar a biblioteca do aplicativo (por exemplo, excluir músicas ou organizá-las em listas de execução).
- Gravar em um CD uma lista de músicas da biblioteca.
- Carregar uma lista de músicas da biblioteca para um iPod ou MP3 player.
- Converter uma música do formato MP3 para o formato AAC e vice-versa.

Essa não é uma lista completa, mas suficiente para entendermos a função dos casos de uso e diagramas de caso de uso.

FIGURA A1.4

Diagrama de distribuição



Um caso de uso descreve como um usuário interage com o sistema definindo os passos necessários para atingir um objetivo específico (por exemplo, gravar uma lista de músicas em um CD). Variações na sequência de passos descrevem vários cenários (por exemplo, o que acontece se as músicas da lista não couberem em um CD?).

Um diagrama UML de caso de uso é uma visão geral de todos os casos de uso e como estão relacionados. Fornece uma visão geral da funcionalidade do sistema. Um diagrama de caso de uso para o aplicativo de música digital é mostrado na Figura A1.5.

Neste diagrama, a figura do usuário representa um *ator* (Capítulo 5) que está associado a uma categoria de usuário (ou outro elemento de interação). Sistemas complexos tipicamente possuem mais de um ator. Por exemplo, um aplicativo “vending machine” pode ter três atores representando clientes, pessoal de manutenção e os fornecedores que abastecem a máquina.

No diagrama de caso de uso, estes são mostrados como ovais. Os atores são conectados por linhas aos casos de uso que eles executam. Note que nenhum dos detalhes dos casos de uso é incluído no diagrama e precisa ser armazenado separadamente. Observe também que os casos de uso são colocados em um retângulo, mas os atores não. Esse retângulo serve para lembrar visualmente as fronteiras do sistema e que os atores estão fora do sistema.

Alguns casos de uso em um sistema podem estar relacionados uns com os outros. Por exemplo, há passos similares para gravar uma lista de músicas para um CD e para carregar uma lista de músicas para um iPod. Em ambos os casos, o usuário primeiro cria uma lista vazia e, em seguida, acrescenta as músicas da biblioteca para a lista. Para evitar duplicação, normalmente é melhor criar um novo caso de uso representando a atividade duplicada, e depois deixar que os outros casos incluam esse novo caso de uso com um de seus passos. A inclusão é indicada nos diagramas de caso de uso, como mostra a Figura A1.6, por meio de uma seta tracejada identificada como «include» conectando um caso de uso a outro.

FIGURA A1.5

Um diagrama de caso de uso para o sistema de música

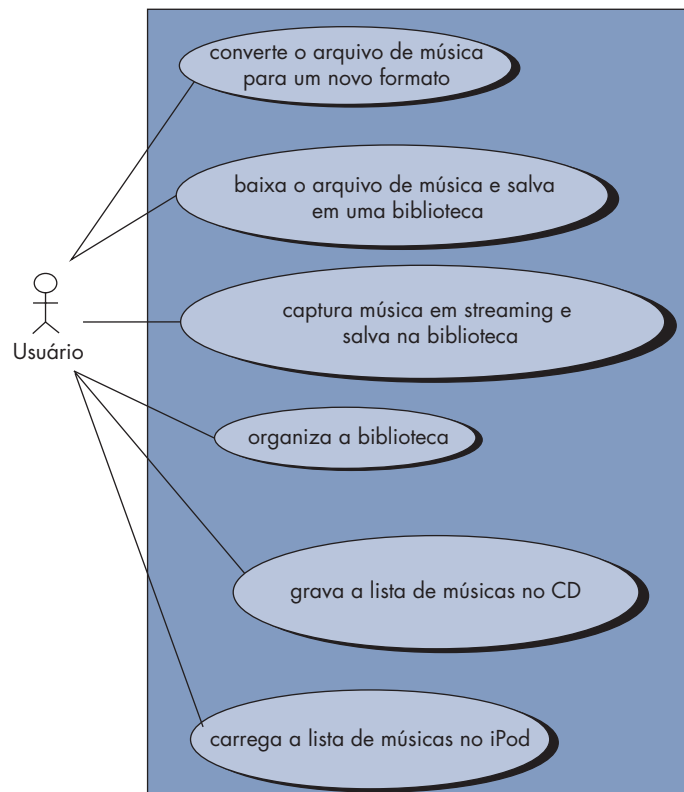
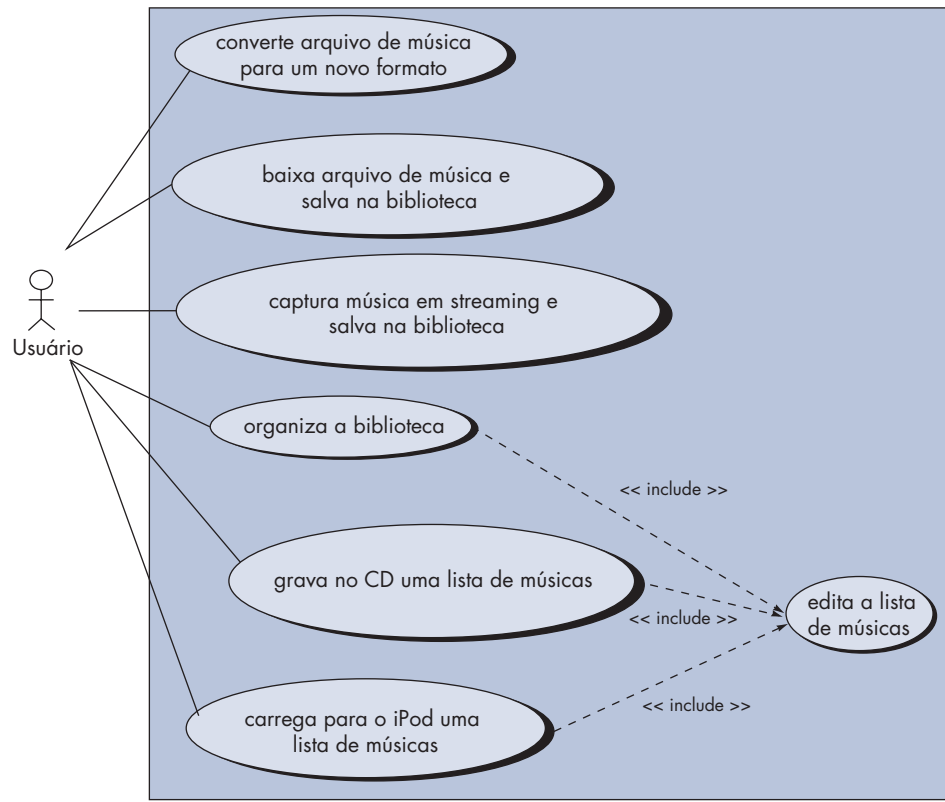


FIGURA A1.6

Um diagrama de caso de uso com casos de uso incluídos



Um diagrama de caso de uso, por mostrar todos os casos, é um bom auxílio para assegurar inclusão de toda a funcionalidade do sistema. No organizador de música digital, certamente você iria querer ter mais casos de uso, como, por exemplo, um para tocar uma música da biblioteca. Mas tenha em mente que a maior contribuição dos casos de uso para o processo de desenvolvimento de software é a descrição textual de cada caso e não o diagrama geral de caso de uso [Fow04b]. É por meio das descrições que você consegue formar uma ideia clara dos objetivos do sistema que está desenvolvendo.

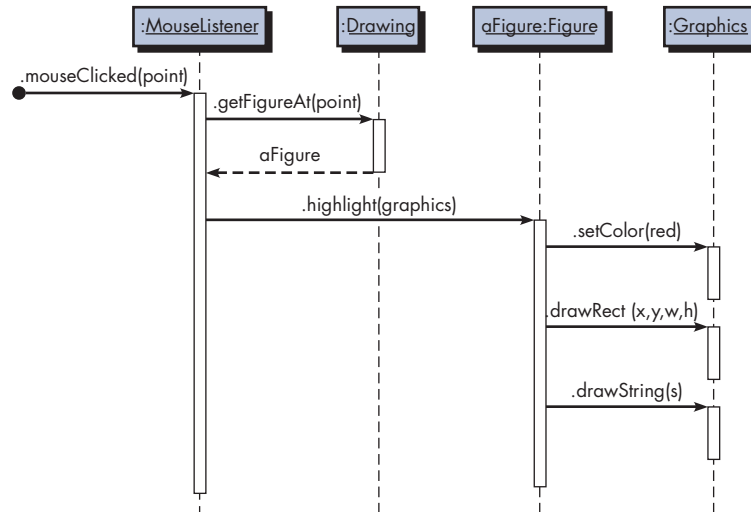
DIAGRAMAS DE SEQUÊNCIA

Em contraste com os diagramas de classe e os de distribuição, que mostram a estrutura estática de um componente de software, o *diagrama de sequência* é utilizado para indicar as comunicações dinâmicas entre objetos durante a execução de uma tarefa. Ele mostra a ordem temporal na qual as mensagens são enviadas entre os objetos para executar aquela tarefa. Podemos usar um diagrama de sequência para mostrar as interações em um caso de uso ou em um cenário de um sistema de software.

Na Figura A1.7, há um diagrama de sequência para um programa de desenho. O diagrama mostra os passos envolvidos para destacar uma figura em um desenho quando é clicada. Cada caixa na linha do topo do diagrama em geral corresponde a um objeto, embora seja possível fazer as caixas modelarem outras coisas, como por exemplo, classes. Se a caixa representa um objeto (como é o caso em todos os nossos exemplos), dentro da caixa pode-se opcionalmente declarar o tipo do objeto precedido de dois-pontos. Você pode também colocar antes dos dois-pontos e do tipo o nome do objeto, conforme mostra a terceira caixa na Figura A1.7. Abaixo de cada caixa há uma linha tracejada chamada de *linha de vida* do objeto. O eixo vertical no diagrama de sequência corresponde ao tempo, e o tempo aumenta à medida que se caminha para baixo.

FIGURA A1.7

Exemplo de um diagrama de sequência



Um diagrama de sequência mostra chamadas de método usando setas horizontais do *chamador* para o *chamado*, identificadas com o nome do método e opcionalmente incluindo seus parâmetros, seus tipos e o tipo de retorno. Por exemplo, na Figura A1.7, **MouseListener** chama o método `getFigureAt()` de **Drawing**. Quando um objeto está executando um método (quando tem uma ativação de um método na pilha), você pode opcionalmente mostrar uma barra branca, conhecida como *barra de ativação*, ao longo da linha de vida do objeto. Na Figura A1.7, há barras de ativação para todas as chamadas de método. O diagrama também pode mostrar opcionalmente o retorno de uma chamada de método com uma seta pontilhada e um rótulo opcional. Na Figura A1.7, o retorno da chamada do método `getFigureAt()` é identificado com o nome do objeto retornado. Uma prática comum, como fizemos na Figura A1.7, é omitir a seta de retorno quando um método não retorna nada (*void*), porque isso complicaria o diagrama fornecendo informações de pouca importância. Um círculo preto com uma seta indica uma *mensagem encontrada* cuja origem é desconhecida ou irrelevante.

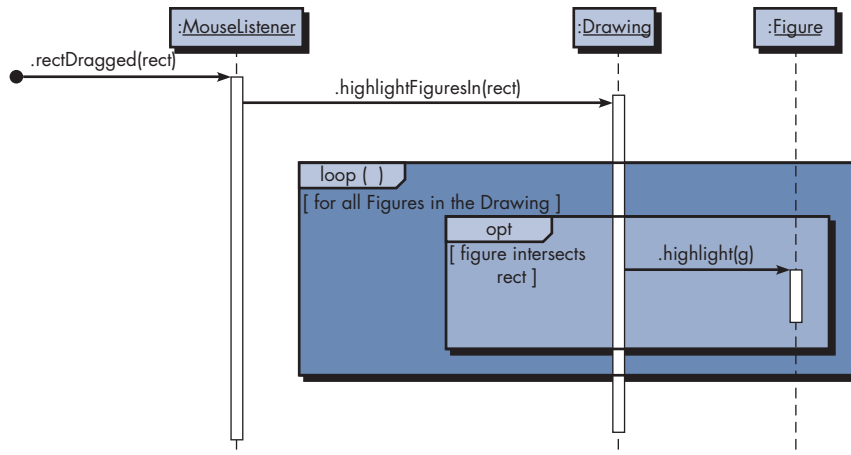
Você será capaz de entender agora a tarefa que a Figura A1.7 está mostrando. Uma origem desconhecida chama o método `mouseClicked()` de um **MouseListener**, passando como argumento o ponto onde o clique ocorreu. O **MouseListener** por sua vez chama o método `getFigureAt()` de um **Drawing**, que retorna um **Figure**. O **MouseListener** então chama o método destacado de **Figure**, passando um objeto **Graphics** como argumento. Em resposta, **Figure** chama três métodos do objeto **Graphics** para traçar a figura em vermelho.

O diagrama na Figura A1.7 é muito claro e não contém condicionais ou laços. Se forem necessárias estruturas lógicas de controle, provavelmente será melhor traçar um diagrama de sequência separado para cada caso. Isto é, se o fluxo de mensagem puder tomar dois caminhos diferentes dependendo de uma condição, trace dois diagramas de sequência separados, um para cada possibilidade.

Se você ainda quer incluir laços, condicionais e outras estruturas de controle em um diagrama de sequência, use frames de interação (*interaction frames*), que são retângulos que envolvem as partes do diagrama e são identificados com o tipo de estruturas de controle que representam. A Figura A1.8 ilustra isso, mostrando o processo envolvido para destacar todas as figuras em um retângulo. Para o **MouseListener** é enviada a mensagem `rectDragged`. O **MouseListener** então manda o desenho destacar todas as figuras no retângulo chamando o método `highlightFigures()`, passando o retângulo como argumento. O método passa em laço por todos os objetos **Figure** no objeto **Drawing** e, se o objeto **Figure** intercepta o retângulo, é solicitado a **Figure** que se destaque. As frases entre colchetes são chamadas de *guardas*, que são condições booleanas que devem ser verdadeiras se a ação dentro da frame de interação deve continuar.

FIGURA A 1.8

Um diagrama de sequência com dois frames de interação



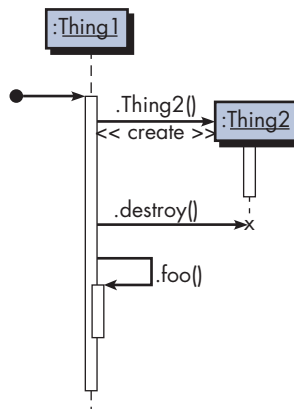
Há muitas outras características que podem ser incluídas em um diagrama de sequência. Por exemplo:

1. Você pode distinguir entre mensagens síncronas e assíncronas. Mensagens síncronas são exibidas com pontas de setas sólidas, enquanto mensagens assíncronas são mostradas com pontas de seta em traço.
2. Você pode mostrar um objeto fazendo-o enviar a si próprio uma mensagem com uma seta saindo do objeto, virando para baixo e, em seguida, apontando de volta para o mesmo objeto.
3. Você pode mostrar a criação do objeto traçando uma seta identificada de forma apropriada (por exemplo, com um rótulo «create») para a caixa de um objeto. Nesse caso, a caixa aparecerá no diagrama abaixo das caixas que correspondem a objetos que já existiam quando a ação começa.
4. Você pode representar a destruição de um objeto com um X grande no fim da linha de vida do objeto. Outros objetos podem destruir um objeto e, nesse caso, uma seta aponta do outro objeto para o X. Um X é útil também para indicar que um objeto não é mais utilizável e está pronto para ser enviado à coleta de lixo.

As três últimas características são mostradas no diagrama de sequência na Figura A1.9.

FIGURA A1.9

Criação, destruição e laços em diagramas de sequência



DIAGRAMAS DE COMUNICAÇÃO

O *diagrama de comunicação* UML (chamado de “diagrama de colaboração” em UML 1.X) fornece outra indicação da ordem temporal das comunicações, mas dá ênfase às relações entre os objetos e classes em vez da ordem temporal.

O diagrama de comunicação, ilustrado na Figura A1.10, exibe as mesmas ações do diagrama de sequência da Figura A1.7.

Em um diagrama de comunicação, os objetos que interagem são representados por retângulos. Associações entre objetos são representadas por linhas ligando os retângulos. Há tipicamente uma seta apontando para um objeto no diagrama, que inicia a sequência de passagem de mensagens. A seta é identificada com um número e um nome de mensagem. Se a mensagem, que chega, for identificada com o número 1 e se ela faz o objeto receptor invocar outras mensagens em outros objetos, aquelas mensagens são representadas por setas do emissor para o receptor com uma linha de associação e recebem números 1.1, 1.2 e assim por diante, na ordem em que são chamadas. Se aquelas mensagens por sua vez invocam outras, é acrescentado outro ponto e outro número ao número que as identifica, para indicar novo aninhamento da mensagem passada.

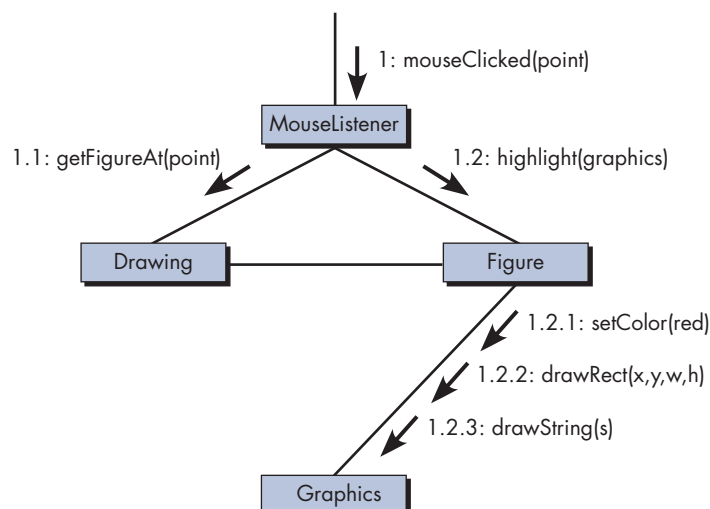
Na Figura A1.10, você vê que a mensagem **mouseClicked** chama os métodos *getFigureAt()* e depois *highlight()*. A mensagem *highlight()* chama três outras mensagens: *setColor()*, *drawRect()* e *drawString()*. A numeração em cada rótulo mostra os aninhamentos, bem como a natureza sequencial de cada mensagem.

Há muitas características opcionais que podem ser acrescentadas aos rótulos das setas. Por exemplo, você pode colocar uma letra na frente do número. Uma seta chegando poderia ser marcada como **A1: mouseClicked(point)**, indicando a execução de uma sequência de comandos (thread), A. Se outras mensagens são executadas em outras threads, o rótulo seria precedido por uma letra diferente. Por exemplo, se o método *mouseClicked()* é executado na thread A, mas ele cria uma nova thread B e chama *highlight()* naquela thread, então a seta de **MouseListener** para **Figure** seria rotulada como **1.B2: highlight(graphics)**.

Se você estiver interessado em mostrar as relações entre os objetos além das mensagens que estão sendo enviadas entre eles, o diagrama de comunicação é provavelmente uma opção melhor do que o diagrama de sequência. Se você estiver mais interessado na ordem temporal da mensagem enviada, o diagrama de sequência provavelmente será melhor.

FIGURA A1.10

Um diagrama de comunicação UML



DIAGRAMAS DE ATIVIDADE

O *diagrama de atividade* mostra o comportamento dinâmico de um sistema ou parte de um sistema através do fluxo de controle entre ações que o sistema executa. Ele é similar a um fluxograma exceto que pode mostrar fluxos concorrentes.

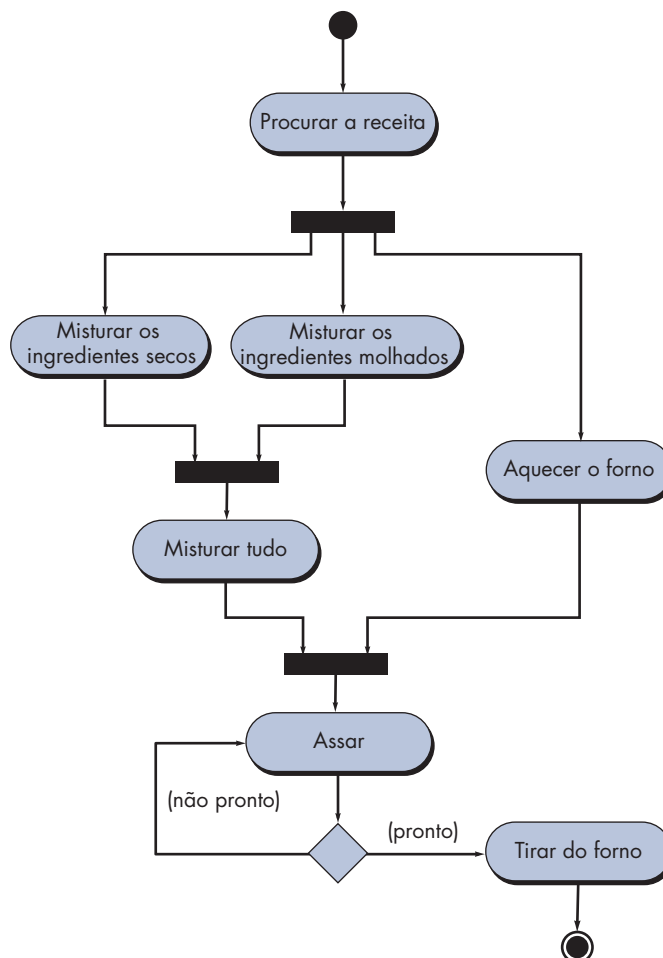
O componente principal de um diagrama de atividade é um *nó ação*, representado por um retângulo arredondado, que corresponde a uma tarefa executada por um sistema de software. Setas que vão de um nó ação para outro indicam o fluxo de controle. Isto é, uma seta entre dois nós ação significa que depois que a primeira ação é completada a segunda ação começa. Um ponto preto sólido forma o *nó inicial* que representa o ponto inicial da atividade. Um ponto preto envolvido por um círculo preto é o *nó final* indicando o fim da atividade.

Um *fork* representa a separação de atividades em duas ou mais atividades concorrentes. Ela é desenhada como uma barra preta horizontal com uma seta apontando para ela e duas ou mais setas apontando para fora dela. Cada seta de saída representa um fluxo de controle que pode ser executado concorrentemente com os fluxos que correspondem a outras setas que saem. Essas atividades concorrentes podem ser executadas em um computador por meio de diferentes threads ou mesmo usando diferentes computadores.

A Figura A1.11 mostra um exemplo de um diagrama de atividade envolvendo a confecção de um bolo. O primeiro passo é procurar a receita. Uma vez encontrada, os ingredientes secos e molhados podem ser medidos e misturados e o forno pode ser pré-aquecido. A mistura dos ingredientes secos pode ser feita em paralelo com a mistura dos ingredientes molhados e com o pré-aquecimento do forno.

FIGURA A1.11

Um diagrama de atividade UML demonstrando como fazer um bolo



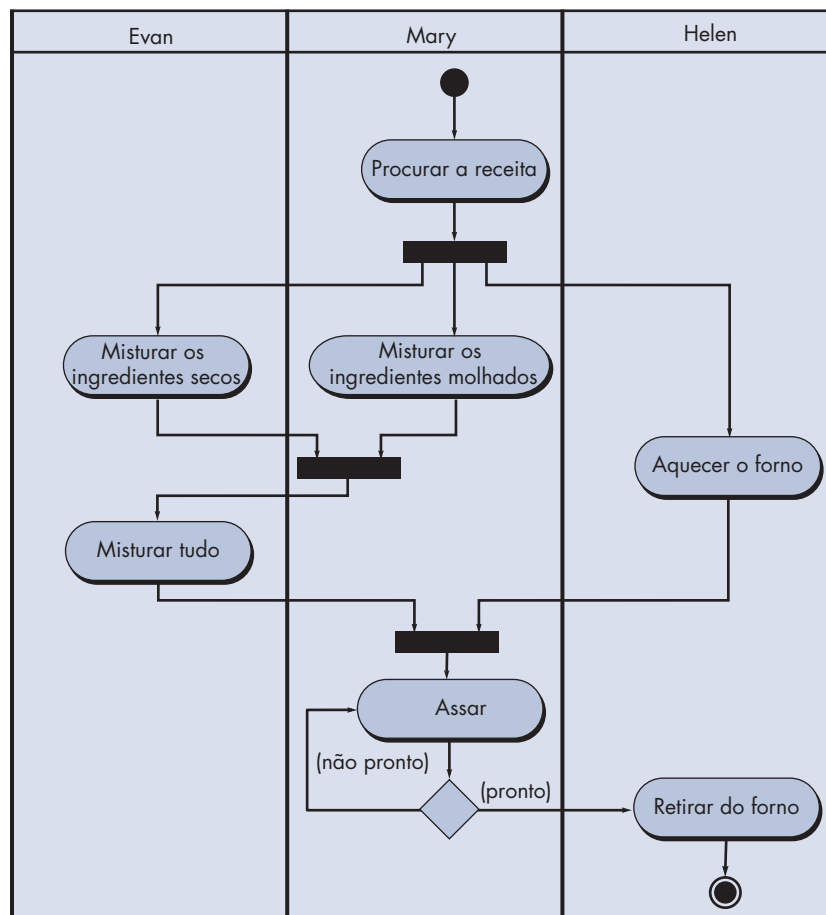
Uma *junção* (*join*) é uma maneira de sincronizar fluxos de controle concorrentes. Ela é representada por uma barra preta horizontal com duas ou mais setas chegando e uma seta saindo. O fluxo de controle representado pela seta que sai não pode iniciar a execução até que todos os outros fluxos representados pelas setas que chegam tenham sido completados. Na Figura A1.11, temos uma junção antes da ação de mistura dos ingredientes secos e molhados. Ela indica que todos os ingredientes secos devem ser misturados e todos os ingredientes molhados devem ser misturados antes que as duas misturas possam ser combinadas. A segunda junção na figura informa que, antes de começar a assar o bolo, todos os ingredientes devem estar misturados e o forno deve estar na temperatura correta.

Um nó *decisão* corresponde a uma ramificação no fluxo de controle baseada em uma condição. Um nó desse tipo é mostrado como um triângulo branco com uma seta chegando e duas ou mais saindo. Cada seta que sai é identificada com uma condição entre colchetes (*guard*). O fluxo de controle segue a seta que sai cuja condição é verdadeira (*true*). É bom certificar-se de que as condições abrangem todas as possibilidades de forma que exatamente uma delas seja verdadeira sempre que for encontrado um nó de decisão. A Figura A1.11 apresenta um nó de decisão após assar o bolo. Se o bolo estiver pronto, ele é removido do forno. Caso contrário, permanece no forno por mais um tempo.

Uma das coisas que o diagrama de atividades da Figura A1.11 não mostra é quem executa cada uma das ações. Muitas vezes, a divisão exata do trabalho não importa. Mas se você quiser indicar como as ações são divididas entre os participantes, decore o diagrama de atividades com raias (*swimlanes*), como mostra a Figura A1.12. As *raias*, como o nome diz, são formadas

FIGURA A1.12

O diagrama de atividades da confecção do bolo com a inclusão de raias



dividindo-se o diagrama em tiras ou “faixas”, e cada uma dessas faixas corresponde a um dos participantes. Todas as ações em uma faixa são executadas pelo participante correspondente. Na Figura A1.12, Evan é responsável pela mistura dos ingredientes secos e depois pela mistura dos ingredientes secos e molhados juntos, Helen é responsável por aquecer o forno e tirar o bolo e Mary por todo o restante.

DIAGRAMAS DE ESTADO

O comportamento de um objeto em determinado instante frequentemente depende do estado do objeto, ou seja, os valores de suas variáveis naquele instante. Como um exemplo trivial, considere um objeto com uma variável de instância booleana. Quando solicitado a executar uma operação, o objeto pode realizar algo se a variável for verdadeira (*true*) e realizar outra coisa se for falsa (*false*).

Um *diagrama de estado* modela os estados de um objeto, as ações executadas dependendo daqueles estados e as transições entre os estados do objeto.

Como exemplo, considere o diagrama de estado para uma parte de um compilador Java. A entrada do compilador é um arquivo de texto, que pode ser considerado uma longa sequência (string) de caracteres. O compilador lê os caracteres, um de cada vez, e a partir deles determina a estrutura do programa. Uma pequena parte desse processo consiste em ignorar caracteres “espaço em branco” (por exemplo, os caracteres *espaço*, *tabulação*, *nova linha*, e *“return”*) e caracteres dentro de um comentário.

Suponha que o compilador delegue ao objeto **WhiteSpaceAndCommentEliminator** a tarefa de avançar sobre os caracteres espaço em branco e caracteres dentro de comentários. A tarefa desse objeto é ler os caracteres de entrada até que todos os espaços em branco e os caracteres entre comentários tenham sido lidos. Nesse ponto ele retorna o controle para o compilador para ler e processar caracteres que não são espaços em branco e não são caracteres de comentários. Pense em como o objeto **WhiteSpaceAndCommentEliminator** lê os caracteres e determina se o próximo caractere é um espaço em branco ou se é parte de um comentário. O objeto pode verificar espaços em branco testando o próximo caractere para saber se é “ ”, “\t”, “\n”, e “\r”. Mas como ele determina se o próximo caractere é parte de um comentário? Por exemplo, quando vê uma “/” pela primeira vez, ele ainda não sabe se aquele caractere representa um operador de divisão, parte do operador /= ou o início de uma linha ou comentário de bloco. Para determinar isso, **WhiteSpaceAndCommentEliminator** precisa registrar o fato de que viu uma “/” e então passar para o próximo caractere. Se o caractere que vem depois da “/” for uma outra “/” ou um “*”, então **WhiteSpaceAndCommentEliminator** saberá que ele está agora lendo um comentário e pode avançar até o fim sem processar ou salvar qualquer caractere. Se o caractere que vem em seguida à primeira “/” for alguma coisa diferente de uma “/” ou um “*”, o objeto **WhiteSpaceAndCommentEliminator** saberá que a “/” representa o operador divisão ou parte do operador /= e assim ele para de avançar sobre os caracteres.

Resumindo, à medida que o objeto **WhiteSpaceAndCommentEliminator** lê os caracteres, ele precisa controlar vários aspectos, incluindo se o caractere atual é um espaço em branco, se o caractere anterior que ele leu era uma “/”, se ele está no momento lendo caracteres de um comentário, se chegou ao fim de um comentário e assim por diante. Tudo isso corresponde a diferentes estados do objeto **WhiteSpaceAndCommentEliminator**. Em cada um desses estados, **WhiteSpaceAndCommentEliminator** se comporta diferentemente com relação ao próximo caractere a ser lido.

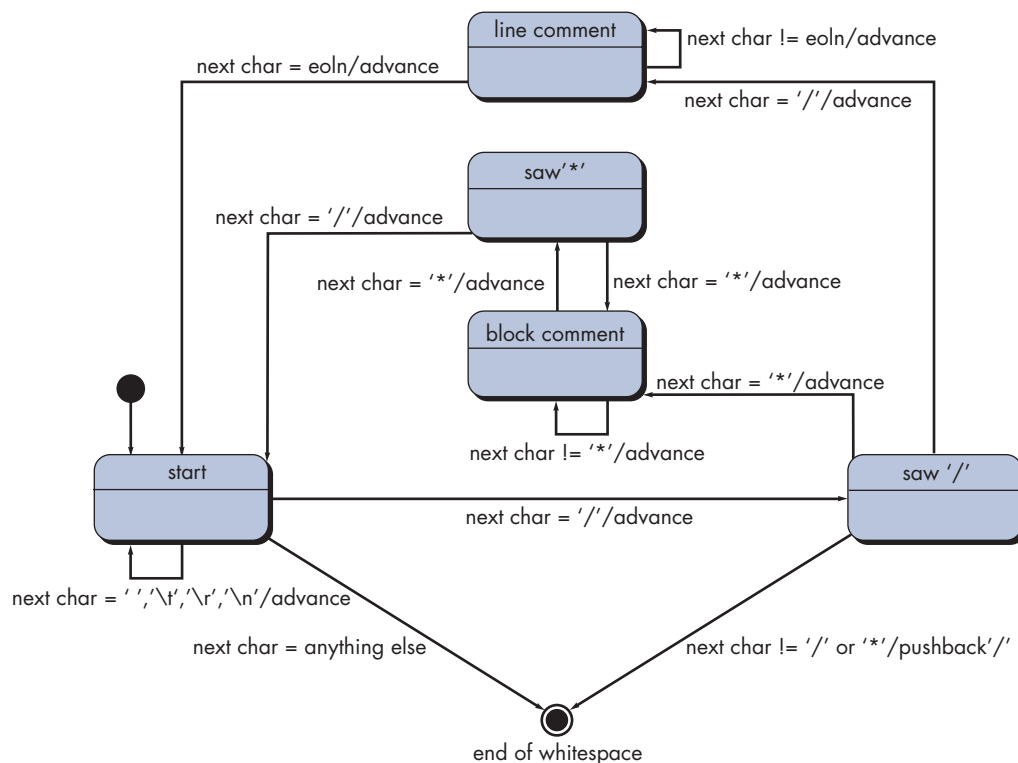
Para ajudar a visualizar todos os estados desse objeto e como ele muda o estado, você pode usar um diagrama de estado conforme indica a Figura A1.13. Um diagrama de estado mostra os estados através de retângulos arredondados, cada um dos quais tem um nome em sua metade superior. Há também um círculo preto chamado de “pseudoestado inicial”, que não é realmente

um estado, mas apenas pontos para o estado inicial. Na Figura A1.13, o estado **start** é o estado inicial. Setas de um estado para outro representam transições ou mudanças no estado do objeto. Cada transição é identificada com um evento disparo, uma barra (/), e uma atividade. Todas as partes dos rótulos de transição são opcionais nos diagramas de estado. Se o objeto está em um estado e ocorre o disparo de evento, para uma de suas transições, a atividade daquela transição é executada e o objeto assume o novo estado indicado pela transição. Por exemplo, na Figura A1.13, se o objeto **WhiteSpaceAndCommentEliminator** estiver no estado **start** e o próximo caractere for `"/'"`, **WhiteSpaceAndCommentEliminator** avança além daquele caractere e muda para o estado **saw '/'**. Se o caractere depois de `"/'"` for outra `"/'"`, o objeto avança para o estado **line comment** e permanece lá até ler um caractere fim de linha. Se por outro lado, o próximo caractere após a `"/'"` for um `"*"`, o objeto avança para o estado **block comment** e permanece lá até encontrar outro `"*"` seguido de uma `"/'"`, que indica o fim de um comentário de bloco. Estude o diagrama para ter certeza de que o entendeu. Note que, após avançar além do espaço em branco ou de um comentário, o objeto **WhiteSpaceAndCommentEliminator** volta para o estado **start** e começa tudo novamente. Esse comportamento é necessário, já que pode haver vários comentários ou caracteres de espaço em branco sucessivos antes de encontrar quaisquer outros caracteres do código-fonte Java.

O objeto pode fazer uma transição para um estado final, indicado por um círculo preto com um círculo branco ao redor dele, que informa que não há mais transições. Na Figura A1.13, o objeto **WhiteSpaceAndCommentEliminator** é encerrado quando o próximo caractere não é um espaço em branco ou parte de um comentário. Note que todas as transições, exceto as duas que levam ao estado final, têm atividades que consistem em avançar para o próximo caractere. As duas transições para o estado final não avançam sobre o próximo caractere porque o pró-

FIGURA A1.13

Um diagrama de estado para avançar além do espaço em branco e comentários em Java



ximo é parte de uma palavra ou símbolo de interesse para o compilador. Note que se o objeto estiver no estado **saw '/'** mas o próximo caractere não é uma "/" ou "*", então o caractere "/" é um operador divisão ou parte do operador /= e então não queremos avançar. Na verdade, queremos retroceder um caractere para tornar a "/" o próximo caractere, para que assim o caractere "/" possa ser usado pelo compilador. Na Figura A1.13, essa atividade de retrocesso é chamada de pushback '/

Um diagrama de estado o ajudará a descobrir situações perdidas ou inesperadas. Com um diagrama de estado, é relativamente fácil garantir que todos os eventos possíveis para todos os estados possíveis foram levados em conta. Na Figura A1.13, você pode facilmente verificar que cada estado tenha incluído transições para todos os caracteres possíveis.

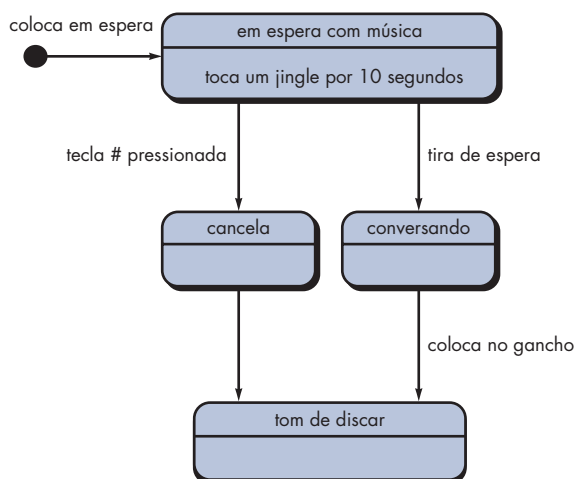
Os diagramas de estado podem conter muitas outras características não incluídas na Figura A1.13. Por exemplo, quando um objeto está em um estado, ele usualmente não faz nada e espera até que ocorra um evento. No entanto, há um tipo especial de estado, denominado *estado de atividade*, no qual o objeto executa alguma atividade, chamada de *do-activity*, enquanto ele está naquele estado. Para indicarmos que um estado é um estado de atividade no diagrama, incluímos na metade inferior do retângulo arredondado de estado a frase "do/" seguida da atividade que deve ser executada enquanto estiver naquele estado. A do-activity pode terminar antes que ocorram quaisquer transições de estado, o estado atividade se comporta como um estado de espera normal. Se ocorrer uma transição fora do estado atividade, antes que a do-activity tenha terminado a do-activity é interrompida.

Devido a um evento ser opcional quando ocorre uma transição, é possível que nenhum evento possa estar listado como parte do rótulo de uma transição. Em casos assim para estados de espera normais, o objeto fará imediatamente uma transição daquele estado para o novo estado. Para estados de atividade, uma transição dessas ocorre logo que do-activity termina.

A Figura A1.14 ilustra essa situação usando os estados para um telefone comercial. Quando uma chamada é colocada em espera, vai para o estado **on hold with music** (música de espera toca por 10 segundos). Após 10 segundos, a do-activity do estado é completada e o estado se comporta como um estado normal de não atividade. Se a pessoa que está chamando pressionar a tecla # quando a chamada está no estado **on hold with music**, a chamada faz uma transição para o estado **canceled**, e o telefone passa imediatamente para o estado **dial tone**. Se a tecla # for pressionada antes de completar os 10 segundos de música de espera, a do-activity é interrompida e a música para imediatamente.

FIGURA A1.14

Um diagrama de estado com um estado de atividade e uma transição "triggerless"



OBJECT CONSTRAINT LANGUAGE – UMA VISÃO GERAL

A grande variedade de diagramas disponíveis como parte da UML proporciona um excelente conjunto de formas de representação para o modelo de projeto. No entanto, as representações gráficas muitas vezes não são suficientes. Você pode precisar de um mecanismo para representar explícita e formalmente informações que restringem alguns elementos do modelo de projeto. É possível, naturalmente, descrever as restrições em uma linguagem natural como o inglês, mas essa abordagem invariavelmente leva a inconsistência e ambiguidade. Por essa razão, parece ser apropriada uma linguagem mais formal – que use a teoria dos conjuntos e linguagens de especificação formal (veja o Capítulo 21), mas tenha de certa forma a sintaxe menos matemática de uma linguagem de programação.

A *Object Constraint Language* (OCL) complementa a UML, permitindo que você use uma gramática e sintaxe formais para construir instruções não ambíguas sobre vários elementos do modelo de projeto (por exemplo, classes e objetos, eventos, mensagens, interfaces). As instruções OCL mais simples são criadas em quatro partes: (1) um *contexto* que define a situação limitada na qual a instrução é válida, (2) uma *propriedade* que representa algumas características do contexto (por exemplo, se o contexto é uma classe, uma propriedade pode ser um atributo), (3) uma *operação* (por exemplo, aritmética, orientada a conjunto) que manipula ou qualifica uma propriedade, e (4) palavras-chave (por exemplo, **if**, **then**, **else**, **and**, **or**, **not**, **implies**) usadas para especificar expressões condicionais.

Como um exemplo simples de uma expressão OCL, considere o sistema de impressão discutido no Capítulo 10. A condição de guarda (*guard condition*) colocada no evento **jobCostAccepted** que causa uma transição entre os estados *computingJobCost* e *formingJob* dentro do diagrama de estado (*statechart diagram*) para o objeto **PrintJob** (Figura 10.9). No diagrama (Figura 10.9), a condição de guarda é expressa em linguagem natural e implica que a autorização só pode ocorrer se o cliente está autorizado para aprovar o custo do trabalho. Em OCL, a expressão pode assumir a forma:

```
customer
    self.authorizationAuthority = 'yes'
```

em que um atributo booleano, **authorizationAuthority**, da classe (na realidade uma instância específica da classe) denominada **Customer** deve ser definida como “yes” para que a condição de guarda seja satisfeita.

Na medida em que o modelo de projeto é criado, muitas vezes há instâncias nas quais pré ou pós-condições devem ser satisfeitas antes de completar alguma ação especificada pelo projeto. A OCL proporciona uma poderosa ferramenta para especificar pré e pós-condições de uma maneira formal. Como exemplo, considere uma ampliação do sistema de impressão (discutido como exemplo no Capítulo 10), no qual o cliente estabelece um limite superior de custo para a impressão e uma data final de entrega no instante em que são especificadas outras características da impressão. Se as estimativas de custo e prazos de entrega exceder esses limites, o trabalho não é aceito e o cliente deve ser notificado. Em OCL, um conjunto de pré e pós-condições pode ser especificado da seguinte maneira:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
    Integer)
pre: upperCostBound > 0
    and custDeliveryReq > 0
    and self.jobAuthorization = 'no'
post: if self.totalJobCost <= upperCostBound
    and self.deliveryDate <= custDeliveryReq
    then
        self.jobAuthorization = 'yes'
    endif
```

Essa instrução OCL define uma invariante (**inv**) – condições que devem existir antes de (pré) e após (pós) algum comportamento. Inicialmente, uma pré-condição estabelece que um limite de custo e de data de entrega deve ser especificado pelo cliente, e a autorização deve ser definida como “no”. Depois que os custos e o prazo de entrega são determinados, é aplicada a pós-condição especificada. Deve-se notar também que para a expressão:

`self.jobAuthorization = 'yes'`

não é atribuído o valor “yes”, mas está declarando que **jobAuthorization** deve ser definido como “yes” quando a operação terminar. Uma descrição completa da OCL está além do escopo deste apêndice. A especificação OCL completa pode ser obtida no site www.omg.org/technology/documents/formal/ocl.htm.

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Há dezenas de livros discutindo UML. Aqueles que tratam da última versão incluem: Miles e Hamilton (*Learning UML 2.0*, O'Reilly Media, Inc., 2006); Booch, Rumbaugh e Jacobson (*Unified Modeling Language User Guide*, 2d ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style*, Cambridge University Press, 2005) e Pilone e Pitman (*UML 2.0 in a Nutshell*, O'Reilly Media, Inc., 2005).

Há disponível na Internet uma ampla variedade de fontes de informação sobre o uso da UML na modelagem de engenharia de software. Uma lista atualizada das referências da Web pode ser encontrada em “análise” e “projeto” no site www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.

CONCEITOS-
-CHAVE

atributos	865
classes	864
características ..	869
controlador	866
definição	863
entidade	866
fronteiras	866
projeto	868
encapsulamento ..	863
herança	866
mensagens	867
métodos	865
operações	865
polimorfismo	868
serviços	865
subclasse	865
superclasse	865

O que é um ponto de vista orientado a objeto? Por que um método é considerado orientado a objeto? O que é um objeto? À medida que os conceitos orientados a objeto ganharam ampla aceitação durante as décadas de 1980 e 1990, surgiram muitas opiniões diferentes sobre as respostas corretas a essas perguntas, mas atualmente há uma visão coerente dos conceitos orientados a objeto. Este apêndice lhe proporcionará uma visão rápida do tópico e apresentará os conceitos básicos e a terminologia.

Para entender o ponto de vista orientado a objeto, considere um exemplo prático – o objeto sobre o qual você está sentado agora – uma cadeira (*chair*). **Chair** é uma subclasse de uma classe muito maior que chamamos de PeçaDeMóvel (**PieceOfFurniture**). Cadeiras individuais são membros (usualmente chamados de instâncias) da classe **Chair**. Um conjunto de atributos genéricos pode ser associado a cada objeto da classe **PieceOfFurniture**. Por exemplo, todos os móveis têm um custo, dimensões, peso, localização e cor, entre muitos outros atributos possíveis. Isso se aplica independentemente de estarmos falando de uma mesa (*table*) ou uma cadeira (*chair*), um sofá (*sofa*) ou um armário (*armoire*). Como **Chair** é um membro de **PieceOfFurniture**, **Chair** herda todos os atributos definidos para a classe.

Tentamos dar uma definição bem-humorada de uma classe descrevendo seus atributos, mas algo está faltando. Todo objeto na classe **PieceOfFurniture** pode ser manipulado de várias maneiras. Ele pode ser comprado e vendido, modificado fisicamente (por exemplo, você pode cortar uma perna da cadeira ou pintar o objeto de púrpura) ou movido de um lugar para outro. Cada uma dessas operações (outros termos são *serviços* ou *métodos*) modificará um ou mais atributos do objeto. Por exemplo, se o atributo localização for um item de dado composto definido como

localização = edifício + piso + sala

uma operação denominada *move()* modificaria um ou mais dos itens de dados (edifício, piso ou sala) que formam o atributo localização. Para isso, *move()* deve ter “conhecimento” desses itens de dados. A operação *move()* poderia ser usada para uma cadeira ou mesa, já que ambas são instâncias da classe **PieceOfFurniture**. Operações válidas para a classe **PieceOfFurniture** – *buy()*, *sell()*, *weigh()* – são especificadas como parte da definição de classe e herdadas por todas as instâncias da classe.

A classe **Chair** (e todos os objetos em geral) encapsula dados (os valores de atributo que definem a cadeira), operações (as ações aplicadas para mudar os atributos da cadeira), outros objetos, constantes (configuração de valores) e outras informações relacionadas. *Encapsulamento* significa que todas essas informações são empacotadas sob um nome e podem ser reutilizadas como uma especificação ou componente de programa.

Agora que introduzimos alguns conceitos básicos, terá mais sentido uma definição formal de *orientado a objeto*. Coad e Yourdon [Coa91] definem o termo da seguinte maneira:

Orientado a objeto = objetos + classificação + herança + comunicação

Três desses conceitos já foram apresentados. Comunicação será discutida mais adiante neste apêndice.

CLASSES E OBJETOS

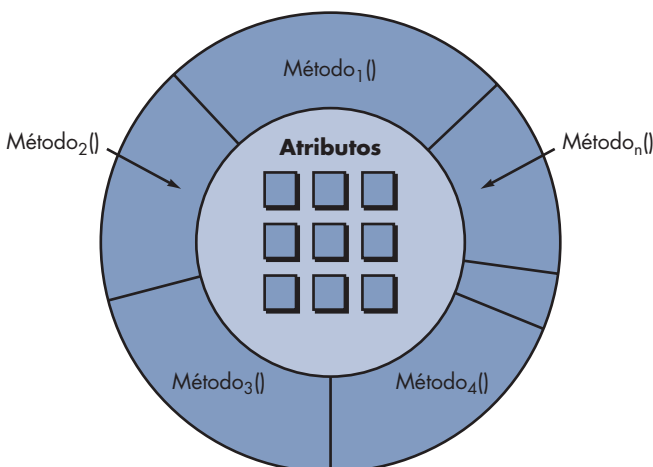
Classe é um conceito orientado a objeto que encapsula dados e abstrações procedurais necessárias para descrever o conteúdo e comportamento de alguma entidade do mundo real. Abstrações de dados que descrevem a classe são envolvidas por uma “parede” de abstrações procedurais [Tay90] (representada na Figura A2.1) capazes de manipular de certa maneira os dados. Em uma classe bem projetada, a única maneira de atingir os atributos (e operar sobre eles) é passar através de um dos métodos que formam a “parede” ilustrada na figura. Portanto, a classe encapsula dados (dentro da parede) e o processamento que manipula os dados (os métodos que formam a parede). Isso possibilita o encapsulamento de informações (Capítulo 8) e reduz o impacto de efeitos colaterais associados a mudança. Como os métodos tendem a manipular um número limitado de atributos, sua coesão é melhorada, e porque a comunicação ocorre somente por meio dos métodos que formam a “parede”, a classe tende a ser menos fortemente acoplada em relação a outros elementos de um sistema.¹

Em outras palavras, podemos dizer que classe é uma descrição generalizada (por exemplo, um *template* ou *blueprint*) que descreve uma coleção de objetos similares. Por definição, objetos são instâncias de uma classe específica e herdam seus atributos e propriedades disponíveis para manipular os atributos. Uma *superclasse* (muitas vezes chamada de *classe base*) é a generalização de um conjunto de classes relacionadas a ela. Uma *subclasse* é uma especialização da superclasse. Por exemplo, a superclasse **MotorVehicle** é uma generalização da classe **Truck**, **SUV**, **Automobile** e **Van**. A subclasse **Automobile** herda todos os atributos de **MotorVehicle**, mas, além disso, incorpora atributos adicionais específicos apenas aos automóveis.

Essas definições implicam a existência de uma hierarquia de classes na qual atributos e operações da superclasse são herdados por subclasses que podem, cada uma delas, acrescentar atributos e métodos “privados” adicionais. Por exemplo, as operações *sitOn()* e *turn()* podem ser privadas à subclasse **Chair**.

FIGURA A2.1

Representação
esquemática de
uma classe



¹ Deve-se notar, no entanto, que o acoplamento pode se tornar um sério problema em sistemas orientados a objeto. Ele surge quando classes de várias partes do sistema são usadas como tipos de dados de atributos e argumentos para métodos. Apesar do acesso aos objetos poder ser apenas por meio de chamadas de procedimento, isso não significa que o acoplamento seja necessariamente baixo, apenas menor do que seria se fosse permitido o acesso direto às partes internas do objeto.

ATRIBUTOS

Você já aprendeu que os atributos são anexados às classes e que as descrevem de certa maneira. O atributo pode assumir um valor definido por um *domínio* enumerado. Em muitos casos, domínio é apenas um conjunto de valores específicos. Suponha que uma classe **Automobile** tenha o atributo cor. O domínio dos valores de cor é {**white, black, silver, gray, blue, red, yellow, green**}. Em situações mais complexas, o domínio pode ser uma classe. Continuando o exemplo, a classe **Automobile** também tem um atributo conjunto de tração (**powerTrain**) que por si só é uma classe. A classe **PowerTrain** teria os atributos que descrevem o motor e a transmissão específicos para determinado carro.

As *características* (valores do domínio) podem ser ampliadas atribuindo-se um valor-padrão a um atributo. Por exemplo, o atributo **cor** por padrão é **white**. Ele pode também ser útil para associar uma probabilidade com determinada característica atribuindo pares {valor, probabilidade}. Considere o atributo **cor** para automóvel. Em algumas aplicações (planejamento de manufatura) pode ser necessário atribuir uma probabilidade a cada uma das cores (branco e preto são altamente prováveis como cores de automóveis).

OPERAÇÕES, MÉTODOS E SERVIÇOS

Um objeto encapsula dados (representados como uma coleção de atributos) e algoritmos que processam os dados. Esses algoritmos são chamados de *operações, métodos* ou *serviços*² e podem ser vistos como componentes de processamento.

Cada uma das operações encapsulada por um objeto proporciona uma representação de um dos comportamentos do objeto. A operação *GetColor()* para o objeto **Automobile** extrairá a cor armazenada no atributo **cor**. A implicação da existência dessa operação é que a classe **Automobile** foi projetada para receber um estímulo (chamamos o estímulo de *mensagem*) que solicita a cor da instância particular de uma classe. Sempre que um objeto recebe um estímulo, ele inicia algum comportamento. Isso pode ser algo simples como obter a cor do automóvel ou complexo como o início de uma cadeia de estímulos passados entre uma variedade de objetos diferentes. Neste último caso, considere um exemplo no qual o estímulo inicial recebido por **Object 1** resulta na geração de dois outros estímulos enviados para **Object 2** e **Object 3**. Operações encapsuladas pelo segundo e terceiro objetos agem sobre o estímulo, retornando informações necessárias para o primeiro objeto. **Object 1** usa as informações retornadas para satisfazer o comportamento demandado pelo estímulo inicial.

CONCEITOS DE ANÁLISE E PROJETO ORIENTADO A OBJETO

A modelagem de requisitos (também chamada de modelagem de análise) concentra-se primeiro sobre classes extraídas diretamente do enunciado do problema. Essas *classes de entidade* tipicamente representam itens que devem ser armazenados em uma base de dados e persistem durante toda a aplicação (a menos que sejam excluídas especificamente).

O projeto refina e amplia o conjunto de classes de entidade. Classes de fronteira e controladoras são desenvolvidas e/ou refinadas durante o projeto. Classes de fronteira criam a interface (por exemplo, telas interativas e relatórios impressos) que o usuário vê e com os quais interage à medida que o software é usado. Classes de fronteira são projetadas com a responsabilidade de controlar a maneira como os objetos entidade são representados para os usuários.

Classes controladoras são projetadas para controlar (1) a criação ou atualização de objetos entidade, (2) a instanciação de objetos de fronteira, já que obtêm informações dos objetos entidade, (3) comunicação complexa entre conjuntos de objetos, e (4) validação de dados transferidos entre objetos ou entre o usuário e a aplicação.

2 No contexto dessa discussão, é usado o termo *operações*, mas os termos *métodos* e *serviços* são igualmente populares.

Os conceitos discutidos nos próximos parágrafos podem ser úteis no trabalho de análise e projeto.

Herança. É um dos diferenciadores-chave entre sistemas convencionais e orientados a objeto. Uma subclasse **Y** herda todos os atributos e operações associadas a sua superclasse **X**. Isso significa que todas as estruturas de dados e algoritmos originalmente desenhados e implementados para **X** ficam imediatamente disponíveis para **Y** – nenhum trabalho adicional precisa ser feito. A reutilização foi conseguida diretamente.

Qualquer alteração nos atributos ou operações contidas dentro de uma superclasse é imediatamente herdada por todas as subclasses. Portanto, a hierarquia de classes torna-se um mecanismo pelo qual alterações (em altos níveis) podem ser imediatamente propagadas através de um sistema.

É importante notar que em cada nível de hierarquia de classe, novos atributos e operações podem ser acrescentados àqueles que foram herdados de níveis mais altos na hierarquia. De fato, sempre que uma nova classe deve ser criada, você tem um conjunto de opções:

- A classe pode ser projetada e construída do início. Isto é, não é usada a herança.
- A hierarquia de classe pode ser pesquisada para determinar se uma classe mais alta na hierarquia contém grande parte dos atributos e operações necessários. A nova classe herda da classe mais alta e podem ser feitas adições quando necessário.
- A hierarquia de classe pode ser reestruturada para que os atributos e operações necessários possam ser herdados pela nova classe.
- As características de uma classe existente podem ser anuladas, e diferentes versões de atributos ou operações são implementadas para a nova classe.

Como todos os conceitos fundamentais de projeto, a herança pode proporcionar benefício significativo ao projeto, mas se ela for usada de forma não apropriada,³ pode complicar um projeto desnecessariamente e levar a um software passível de erros difícil de manter.

Mensagens. As classes devem interagir umas com as outras para atingir os objetivos do projeto. Uma mensagem estimula a ocorrência de algum comportamento no objeto receptor. O comportamento ocorre quando uma operação é executada.

A interação entre objetos é apresentada na Figura A2.2. Uma operação dentro de **SenderObject** gera uma mensagem da forma *mensagem* (<parâmetros>) em que os parâmetros identificam **ReceiverObject** como o objeto a ser estimulado pela mensagem, a operação dentro de **ReceiverObject** que deve receber a mensagem e os itens de dados que fornecem as informações necessárias para que a operação seja bem-sucedida. A colaboração definida entre classes como parte do modelo de requisitos fornece diretrizes úteis na criação das mensagens.

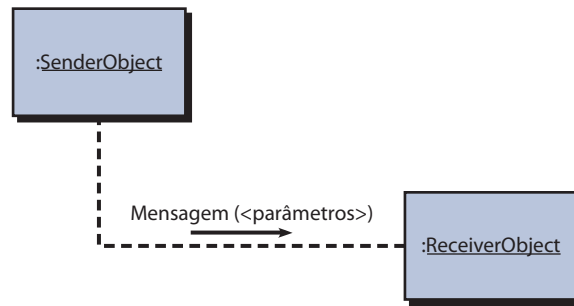
Cox [Cox86] descreve o intercâmbio entre classes da seguinte maneira:

É solicitado a um objeto [classe] que execute uma de suas operações enviando-lhe uma mensagem com as informações do que fazer. O receptor [objeto] responde a mensagem primeiro escolhendo a operação que implementa o nome da mensagem, executando essa operação e, então, retornando o controle para o chamador. As mensagens unem um sistema orientado a objeto. As mensagens proporcionam informações sobre o comportamento dos objetos individuais e do sistema orientado a objeto como um todo.

Polimorfismo. Característica que reduz bastante o esforço necessário para ampliar o projeto de um sistema orientado a objeto. Para entender o polimorfismo, considere uma aplicação convencional que deve traçar quatro tipos diferentes de gráficos: gráficos de colunas, gráficos de pizza, histogramas e diagramas Kiviat. Idealmente, uma vez coletados os dados para um tipo particular de gráfico, o gráfico será traçado. Para conseguir isso em uma aplicação convencional

³ Projetar uma subclasse que herda atributos e operações de mais de uma superclasse (às vezes chamada de “herança múltipla”) não é visto com simpatia por muitos projetistas.

FIGURA A2.2

Mensagens
entre objetos

(e manter a coesão do módulo), seria necessário desenvolver módulos de desenho para cada tipo de gráfico. Então, no projeto, estaria embutida lógica de controle similar a esta a seguir:

```

case of graphtype:
    if graphtype = linegraph then DrawLineGraph (data);
    if graphtype = piechart then DrawPieChart (data);
    if graphtype = histogram then DrawHisto (data);
    if graphtype = kiviatic then DrawKiviatic (data);
end case;
  
```

Embora esse projeto seja razoavelmente simples, seria complicado adicionar novos tipos de gráficos. Um novo módulo de desenho teria de ser criado para cada tipo de gráfico e a lógica de controle teria de ser atualizada para refletir o novo tipo de gráfico.

Para resolver esse problema em um sistema orientado a objeto, todos os gráficos se tornam subclasses de uma classe geral denominada **Graph**. Usando um conceito chamado *sobrecarga* [Tay90], cada subclasse define uma operação *draw*. O objeto pode enviar uma mensagem *draw* a qualquer um dos objetos instanciados a partir de qualquer uma das subclasses. O objeto que está recebendo a mensagem chamará sua própria operação *draw* para criar o gráfico apropriado. Portanto, o projeto é reduzido a

```
draw <graphtype>
```

Quando um novo tipo de gráfico é acrescentado ao sistema, cria-se uma subclasse com sua própria operação *draw*. Mas não são necessárias alterações em qualquer objeto que queira que um gráfico seja desenhado, pois a mensagem **draw <graphtype> permanece inalterada**. Resumindo, o polimorfismo permite que várias operações diferentes tenham o mesmo nome. Isso, por sua vez, desacopla os objetos uns dos outros, tornando-os mais independentes.

Classes de Projeto. O modelo de requisitos define um conjunto completo de classes de análise. Cada uma descreve algum elemento do domínio do problema, focalizando os aspectos visíveis ao usuário ou ao cliente. O nível de abstração de uma classe de análise é relativamente alto.

Conforme evolui o modelo de projeto, a equipe de software deve definir um conjunto de *classes de projeto* que (1) refina as *classes de análise*, fornecendo detalhe de projeto que permitirá que sejam implementadas e (2) crie um novo conjunto de classes de projeto que implemente uma infraestrutura de software que suporte a solução de negócio. São sugeridos cinco tipos diferentes de classes de projeto, cada um representando uma camada diferente da arquitetura de projeto [Amb01]:

- *Classes de interface de usuário* definem todas as abstrações necessárias para a interação homem-computador (*human-computer interaction* – HCI).
- *Classes do domínio de negócio* são muitas vezes refinamentos das classes de análise definidas anteriormente. As classes identificam os atributos e operações (métodos) necessários para implementar algum elemento do domínio de negócio.

- *Classes de processo* implementam abstrações de negócio de baixo nível necessárias para gerenciar completamente as classes do domínio de negócios.
- *Classes persistentes* representam armazenamento de dados (por exemplo, uma base de dados) que persistirá além da execução do software.
- *Classes de sistema* implementam funções de gerenciamento e controle de software que permitem ao sistema operar e comunicar-se em seu ambiente de computação e com o mundo exterior.

À medida que evolui o projeto arquitetural, a equipe de software deverá desenvolver um conjunto completo de atributos e operações para cada classe de projeto. O nível de abstração é reduzido conforme cada classe de análise é transformada em uma representação de projeto. Isto é, classes de análise representam objetos (e métodos associados aplicados a eles) usando o jargão do domínio de negócio. Classes de projeto apresentam um detalhe significativamente mais técnico como um guia para a implementação.

Arlow e Neustadt [Arl02] sugerem que cada classe de projeto seja revista para garantir sua “boa formação”. Eles definem quatro características de uma classe de projeto bem formada:

Completa e suficiente. Uma classe de projeto deverá ser o encapsulamento completo de todos os atributos e métodos requeridos a uma classe (com base em uma interpretação reconhecível do nome da classe). Por exemplo, a classe **Scene** definida para software de edição de vídeo é completa apenas se contiver todos os atributos e métodos que podem ser razoavelmente associados à criação de uma cena de vídeo. A suficiência garante que a classe de projeto contenha somente os métodos necessários para atingir a finalidade da classe, nem mais nem menos.

Primitivismo. Os métodos associados a uma classe de projeto devem se concentrar na execução de uma função específica para a classe. Uma vez que a função tenha sido implementada com um método, a classe não deve proporcionar nenhuma outra maneira de fazê-lo. Por exemplo, a classe **VideoClip** do software de edição de vídeo pode ter atributos **start-point** e **end-point** para indicar os pontos de início e fim do videoclipe (note que o vídeo “bruto” carregado no sistema pode ser maior do que o videoclipe usado). Os métodos, *setStartPoint()* e *setEndPoint()* proporcionam o único meio para estabelecer pontos de início e fim para o videoclipe.

Alta coesão. Uma classe de projeto coesa é limitada. Ela tem um conjunto de responsabilidades pequeno e concentrado e aplica de forma simples atributos e métodos para implementar aquelas responsabilidades. Por exemplo, a classe **VideoClip** do software de edição de vídeo pode conter um conjunto de métodos para editar o videoclipe. Contanto que cada método se concentre somente em atributos associados ao videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário que as classes de projeto colaborem umas com as outras. No entanto, a colaboração deverá ser mantida em um nível mínimo aceitável. Se um modelo de projeto é altamente acoplado (todas as classes de projeto colaboram com todas as outras classes de projeto), o sistema é difícil de implementar, testar e manter com o decorrer do tempo. Em geral, classes de projeto em um subsistema deverão ter apenas um limitado conhecimento das outras classes. Essa restrição, chamada de *Lei de Demeter* [Lie03], sugere que um método deverá mandar mensagens apenas para métodos em classes vizinhas.⁴

LEITURAS E FONTES DE INFORMAÇÃO COMPLEMENTARES

Durante as últimas três décadas, centenas de livros foram escritos sobre programação orientada a objeto, análise e projeto. Weisfeld (*The Object-Oriented Thought Process*, 2d ed., Sams

⁴ Uma maneira menos formal de enunciar a Lei de Demeter é: “Cada unidade deve falar somente com seus amigos; não deve falar com estranhos.”

Publishing, 2003) apresenta um excelente tratamento dos conceitos e princípios gerais orientados a objeto. McLaughlin e seus colegas (*Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*, O'Reilly Media, Inc., 2006) proporcionam um tratamento acessível e leve sobre abordagens de análise e projeto orientado a objeto. Um tratamento mais aprofundado sobre análise e projeto orientado a objeto é apresentado por Booch e seus colegas (*Object-Oriented Analysis and Design with Applications*, 3d ed., Addison-Wesley, 2007). Wu (*An Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2005) escreveu um livro esclarecedor sobre programação orientada a objeto, típico de dezenas de outras publicações escritas para muitas linguagens diferentes.

Uma grande variedade de fontes de informação sobre tecnologias orientadas a objeto está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “análise” e “design” no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

- [Abb83] Abbott, R., "Program Design by Informal English Descriptions," *CACM*, vol. 26, no. 11, November 1983, pp. 892–894.
- [ACM98] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 1998, available at www.acm.org/serving/se/code.htm.
- [Ada93] Adams, D., *Mostly Harmless*, Macmillan, 1993.
- [AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-45, U.S. Air Force, September 30, 1988.
- [Agi03] The Agile Alliance Home Page, www.agilealliance.org/home.
- [Air99] Airline Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.
- [Aka04] Akao, Y., *Quality Function Deployment*, Productivity Press, 2004.
- [Ale77] Alexander, C., *A Pattern Language*, Oxford University Press, 1977.
- [Ale79] Alexander, C., *The Timeless Way of Building*, Oxford University Press, 1979.
- [Amb95] Ambler, S., "Using Use-Cases," *Software Development*, July 1995, pp. 53–61.
- [Amb98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1998.
- [Amb01] Ambler, S., *The Object Primer*, 2d ed., Cambridge University Press, 2001.
- [Amb02a] Ambler, S., "What Is Agile Modeling (AM)?" 2002, www.agilemodeling.com/index.htm.
- [Amb02b] Ambler, S., and R. Jeffries, *Agile Modeling*, Wiley, 2002.
- [Amb02c] Ambler, S., "UML Component Diagramming Guidelines," available at www.modelingstyle.info/, 2002.
- [Amb04] Ambler, S., "Examining the Cost of Change Curve," in *The Object Primer*, 3d ed., Cambridge University Press, 2004.
- [Amb06] Ambler, S., "The Agile Unified Process (AUP), 2006, available at www.ambysoft.com/unifiedprocess/agileUP.html.
- [And06] Andrews, M., and J. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley, 2006.
- [ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.
- [Ant06] Anton, D., and C. Anton, *ISO 9001 Survival Guide*, 3d ed., AEM Consulting Group, 2006.
- [AOS07] AOSD.net (Aspect-Oriented Software Development), glossary, available at <http://aosd.net/wiki/index.php?title=Glossary>.
- [App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," February 2000, available at www.cmcrossroads.com/bradapp/docs/patterns-intro.html.
- [App08] Apple Computer, *Accessibility*, 2008, available at www.apple.com/disability/.
- [Arl02] Arlow, J., and I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002.
- [Arn89] Arnold, R. S., "Software Restructuring," *Proc. IEEE*, vol. 77, no. 4, April 1989, pp. 607–617.
- [Art97] Arthur, L. J., "Quantum Improvements in Software System Quality," *CACM*, vol. 40, no. 6, June 1997, pp. 47–52.
- [Ast04] Astels, D., *Test Driven Development: A Practical Guide*, Prentice Hall, 2004.
- [Ave04] Aversan, L., et al., "Managing Coordination and Cooperation in Distributed Software

- Processes: The GENESIS Environment," *Software Process Improvement and Practice*, vol. 9, Wiley Interscience, 2004, pp. 239–263.
- [Baa07] de Baar, B., "Project Risk Checklist," 2007, available at www.softwareprojects.org/project_riskmanagement_starting62.htm.
- [Bab86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.
- [Bac97] Bach, J., "Good Enough Quality: Beyond the Buzzword," *IEEE Computer*, vol. 30, no. 8, August 1997, pp. 96–98.
- [Bac98] Bach, J., "The Highs and Lows of Change Control," *Computer*, vol. 31, no. 8, August 1998, pp. 113–115.
- [Bae98] Baetjer, Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85.
- [Bak72] Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, vol. 11, no. 1, 1972, pp. 56–73.
- [Ban06] Baniassad, E., et al., "Discovering Early Aspects," *IEEE Software*, vol. 23, no. 1, January–February, 2006, pp. 61–69.
- [Bar06] Baresi, L., E. DiNitto, and C. Ghezzi, "Toward Open-World Software: Issues and Challenges," *IEEE Computer*, vol. 39, no. 10, October 2006, pp. 36–43.
- [Bas84] Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering*, vol. SE-10, 1984, pp. 728–738.
- [Bas03] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, 2d ed., Addison-Wesley, 2003.
- [Bec00] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
- [Bec01a] Beck, K., et al., "Manifesto for Agile Software Development," www.agilemanifesto.org/.
- [Bec04a] Beck, K., *Extreme Programming Explained: Embrace Change*, 2d ed., Addison-Wesley, 2004.
- [Bec04b] Beck, K., *Test-Driven Development: By Example*, 2d ed., Addison-Wesley, 2002.
- [Bee99] Beedle, M., et al., "SCRUM: An Extension Pattern Language for Hyperproductive Software Development," included in: *Pattern Languages of Program Design 4*, Addison-Wesley Longman, Reading MA, 1999, downloadable from http://jeffsutherland.com/scrum/scrum_plop.pdf.
- [Bei84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold, 1984.
- [Bei90] Beizer, B., *Software Testing Techniques*, 2d ed., Van Nostrand-Reinhold, 1990.
- [Bei95] Beizer, B., *Black-Box Testing*, Wiley, 1995.
- [Bel81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L. J. Peters, author), Yourdon Press, 1981.
- [Bel95] Bellinzona R., M. G. Gugini, and B. Pernici, "Reusing Specifications in OO Applications," *IEEE Software*, March 1995, pp. 65–75.
- [Ben99] Bentley, J., *Programming Pearls*, 2d ed., Addison-Wesley, 1999.
- [Ben00] Bennatan, E. M., *Software Project Management: A Practitioner's Approach*, 3d ed., McGraw-Hill, 2000.
- [Ben02] Bennett, S., S. McRobb, and R. Farmer, *Object-Oriented Analysis and Design*, 2d ed., McGraw-Hill, 2002.
- [Ber80] Bersoff, E. H., V. D. Henderson, and S. G. Siegel, *Software Configuration Management*, Prentice Hall, 1980.
- [Ber93] Berard, E., *Essays on Object-Oriented Software Engineering*, vol. 1, Addison-Wesley, 1993.
- [Bes04] Bessin, J., "The Business Value of Quality," IBM developerWorks, June 15, 2004, available at www-128.ibm.com/developerworks/rational/library/4995.html.

- [Bha06] Bhat, J., M. Gupta, and S. Murthy, "Lessons from Offshore Outsourcing," *IEEE Software*, vol. 23, no. 5, September–October 2006.
- [Bie94] Bieman, J. M., and L. M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Engineering*, vol. SE-20, no. 8, August 1994, pp. 308–320.
- [Bin93] Binder, R., "Design for Reuse Is for Real," *American Programmer*, vol. 6, no. 8, August 1993, pp. 30–37.
- [Bin94a] Binder, R., "Testing Object-Oriented Systems: A Status Report," *American Programmer*, vol. 7, no. 4, April 1994, pp. 23–28.
- [Bin94b] Binder, R. V., "Object-Oriented Software Testing," *Communications of the ACM*, vol. 37, no. 9, September 1994, p. 29.
- [Bin99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [Bir98] Biró, M., and T. Remzső, "Business Motivations for Software Process Improvement," ER-CIM News No. 32, January 1998, available at www.ercim.org/publication/Ercim_News/enw32/biro.html.
- [Boe81] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.
- [Boe88] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer*, vol. 21, no. 5, May 1988, pp. 61–72.
- [Boe89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.
- [Boe96] Boehm, B., "Anchoring the Software Process," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 73–82.
- [Boe98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer*, vol. 31, no. 7, July 1998, pp. 33–44.
- [Boe00] Boehm, B., et al., *Software Cost Estimation in COCOMO II*, Prentice Hall, 2000.
- [Boe01a] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition," *Crosstalk*, May 2001, available at www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.
- [Boe01b] Boehm, B., and V. Basili, "Software Defect Reduction Top 10 List," *IEEE Computer*, vol. 34, no. 1, January 2001, pp. 135–137.
- [Boe08] Boehm, B., "Making a Difference in the Software Century," *IEEE Computer*, vol. 41, no. 3, March 2008, pp. 32–38.
- [Boh66] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *CACM*, vol. 9, no. 5, May 1966, pp. 366–371.
- [Boh00] Bohl, M., and M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic*, 5th ed., Prentice Hall, 2000.
- [Boi04] Boiko, B., *Content Management Bible*, 2d ed., Wiley, 2004.
- [Bol02] Boldyreff, C., et al., "Environments to Support Collaborative Software Engineering," 2002, downloadable from www.cs.put.poznan.pl/dweiss/site/publications/download/csmrepaper.pdf.
- [Boo94] Booch, G., *Object-Oriented Analysis and Design*, 2d ed., Benjamin Cummings, 1994.
- [Boo05] Booch, G., J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide*, 2d ed., Addison-Wesley, 2005.
- [Boo06] Bootstrap-institute.com, 2006, www.cse.dcu.ie/espinode/directory/directory.html.
- [Boo08] Booch, G., *Handbook of Software Architecture*, 2008, available at www.booch.com/architecture/systems.jsp.
- [Bor01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.
- [Bos00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.
- [Bra85] Bradley, J. H., "The Science and Art of Debugging," *Computerworld*, August 19, 1985, pp. 35–38.

- [Bra94] Bradac, M., D. Perry, and L. Votta, "Prototyping a Process Monitoring Experiment," *IEEE Trans. Software Engineering*, vol. 20, no. 10, October 1994, pp. 774–784.
- [Bre02] Breen, P., "Exposing the Fallacy of 'Good Enough' Software," informit.com, February 1, 2002, available at www.informit.com/articles/article.asp?p=25141&rl=1.
- [Bro95] Brooks, F., *The Mythical Man-Month*, Silver Anniversary edition, Addison-Wesley, 1995.
- [Bro96] Brown, A. W., and K. C. Wallnau, "Engineering of Component Based Systems," *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7–15.
- [Bro01] Brown, B., *Oracle9i Web Development*, 2d ed., McGraw-Hill, 2001.
- [Bro03] Brooks, F., "Three Great Challenges for Half-Century-Old Computer Science," *JACM*, vol. 50, no. 1, January 2003, pp. 25–26.
- [Bro06] Broy, M., "The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems," *IEEE Computer*, vol. 39, no. 10, October 2006, pp. 72–80.
- [Buc99] Bucanac, C., "The V-Model," University of Karlskrona/Ronneby, January 1999, downloadable from www.bucanac.com/documents/The_V-Model.pdf.
- [Bud96] Budd, T., *An Introduction to Object-Oriented Programming*, 2d ed., Addison-Wesley, 1996.
- [Bus96] Buschmann, F., et al., *Pattern-Oriented Software Architecture*, Wiley, 1996.
- [Bus07] Buschmann, F., et al., *Pattern-Oriented Software Architecture, A System of Patterns*, Wiley, 2007.
- [Cac02] Cachero, C., et al., "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification," *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, June 2002, downloadable from www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.
- [Cai03] Caine, Frarber, and Gordon, Inc., *PDL/81*, 2003, available at www.cfg.com/pdl81/lpd.html.
- [Car90] Card, D. N., and R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.
- [Cas89] Cashman, M., "Object Oriented Domain Analysis," *ACM Software Engineering Notes*, vol. 14, no. 6, October 1989, p. 67.
- [Cav78] Cavano, J. P., and J. A. McCall, "A Framework for the Measurement of Software Quality," *Proc. ACM Software Quality Assurance Workshop*, November 1978, pp. 133–139.
- [CCS02] CS3 Consulting Services, 2002, www.cs3inc.com/DSDM.htm.
- [Cec06] Cechich, A., et al., "Trends on COTS Component Identification," *Proc. Fifth Intl. Conf. on COTS-Based Software Systems*, IEEE, 2006.
- [Cha89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/Intertext, 1989.
- [Cha92] Charette, R. N., "Building Bridges over Intelligent Rivers," *American Programmer*, vol. 5, no. 7, September 1992, pp. 2–9.
- [Cha93] de Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.
- [Cha03] Chakravarti, A., "Online Software Design Pattern Links," 2003, available at www.anupriyo.com/oopfm.shtml.
- [Che77] Chen, P., *The Entity-Relationship Approach to Logical Database Design*, QED Information Systems, 1977.
- [Chi94] Chidamber, S. R., and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 476–493.
- [Cho89] Choi, S. C., and W. Scacchi, "Assuring the Correctness of a Configured Software Description," *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, October 1989, pp. 66–75.
- [Chu95] Churcher, N. I., and M. J. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 69–76.

- [Cig07] Cigital, Inc., "Case Study: Finding Defects Earlier Yields Enormous Savings," 2007, available at www.cigital.com/solutions/roi-cs2.php.
- [Cla05] Clark, S., and E. Baniasaad, *Aspect-Oriented Analysis and Design*, Addison-Wesley, 2005.
- [Cle95] Clements, P., "From Subroutines to Subsystems: Component Based Software Development," *American Programmer*, vol. 8, no. 11, November 1995.
- [Cle03] Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2003.
- [Cle06] Clemmons, R., "Project Estimation with Use Case Points," *CrossTalk*, February 2006, p. 18–222, downloadable from www.stsc.hill.af.mil/crosstalk/2006/02/0602Clemmons.pdf.
- [CMM07] *Capability Maturity Model Integration (CMMI)*, Software Engineering Institute, 2007, available at www.sei.cmu.edu/cmmi/.
- [CMM08] *People Capability Maturity Model Integration (People CMM)*, Software Engineering Institute, 2008, available at www.sei.cmu.edu/cmm-p/.
- [Coa91] Coad, P., and E. Yourdon, *Object-Oriented Analysis*, 2d ed., Prentice Hall, 1991.
- [Coa99] Coad, P., E. Lefebvre, and J. DeLuca, *Java Modeling in Color with UML*, Prentice Hall, 1999.
- [Coc01a] Cockburn, A., and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer*, vol. 34, no. 11, November 2001, pp. 131–133.
- [Coc01b] Cockburn, A., *Writing Effective Use-Cases*, Addison-Wesley, 2001.
- [Coc02] Cockburn, A., *Agile Software Development*, Addison-Wesley, 2002.
- [Coc04] Cockburn, A., "What the Agile Toolbox Contains," *CrossTalk*, November 2004, available at www.stsc.hill.af.mil/crosstalk/2004/11/0411Cockburn.html.
- [Coc05] Cockburn, A., *Crystal Clear*, Addison-Wesley, 2005.
- [Con96] Conradi, R., "Software Process Improvement: Why We Need SPIQ," NTNU, October 1996, downloadable from www.idi.ntnu.no/grupper/su/publ/pdf/nik96-spiq.pdf.
- [Con02] Conradi, R., and A. Fuggetta, "Improving Software Process Improvement," *IEEE Software*, July–August 2002, pp. 2–9, downloadable from <http://citeseer.ist.psu.edu/conradi02improving.html>.
- [Con93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization, *CACM*, vol. 36, no. 10, October 1993, pp. 34–43.
- [Con95] Constantine, L., "What DO Users Want? Engineering Usability in Software," *Windows Tech Journal*, December 1995, available from www.forUse.com.
- [Con03] Constantine, L., and L. Lockwood, *Software for Use*, Addison-Wesley, 1999; see also www.foruse.com/.
- [Cop05] Coplien, J., "Software Patterns," 2005, available at <http://hillside.net/patterns/definition.html>.
- [Cor98] Corfman, R., "An Overview of Patterns," in *The Patterns Handbook*, SIGS Books, 1998.
- [Cou00] Coulouris, G., J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3d ed., Addison-Wesley, 2000.
- [Cox86] Cox, Brad, *Object-Oriented Programming*, Addison-Wesley, 1986.
- [Cri92] Christel, M. G., and K. C. Kang, "Issues in Requirements Elicitation," Software Engineering Institute, CMU/SEI-92-TR-12 7, September 1992.
- [Cro79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.
- [Cro07] Cross, M., and M. Fisher, *Developer's Guide to Web Application Security*, Syngress Publishing, 2007.
- [Cur86] Curritt, P. A., M. Dyer, and H. D. Mills, "Certifying the Reliability of Software," *IEEE Trans, Software Engineering*, vol. SE-12, no. 1, January 1994.

- [Cur88] Curtis, B., et al., "A Field Study of the Software Design Process for Large Systems," *IEEE Trans. Software Engineering*, vol. SE-31, no. 11, November 1988, pp. 1268–1287.
- [Cur01] Curtis, B., W. Hefley, and S. Miller, *People Capability Maturity Model*, Addison-Wesley, 2001.
- [CVS07] Concurrent Versions System, Ximbiot, http://ximbiot.com/cvs/wiki/index.php?title=Main_Page, 2007.
- [DAC03] "An Overview of Model-Based Testing for Software," Data and Analysis Center for Software, CR/TA 12, June 2003, downloadable from www.goldpractices.com/download/practice/pdf/Model_Based_Testing.pdf.
- [Dah72] Dahl, O., E. Dijkstra, and C. Hoare, *Structured Programming*, Academic Press, 1972.
- [Dar91] Dart, S., "Concepts in Configuration Management Systems," *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, downloadable from www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.
- [Dar99] Dart, S., "Change Management: Containing the Web Crisis," *Proc. Software Configuration Management Symposium*, Toulouse, France, 1999, available at www.perforce.com/perforce/conf99/dart.html.
- [Dar01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, available at www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.
- [Das05] Dasari, R., "Lean Software Development," a white paper, downloadable from www.projectperfect.com.au/downloads/Info/info_lean_development.pdf, 2005.
- [Dav90] Davenport, T. H., and J. E. Young, "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Review*, Summer 1990, pp. 11–27.
- [Dav93] Davis, A., et al., "Identifying and Measuring Quality in a Software Requirements Specification," *Proc. First Intl. Software Metrics Symposium*, IEEE, Baltimore, MD, May 1993, pp. 141–152.
- [Dav95a] Davis, M., "Process and Product: Dichotomy or Duality," *Software Engineering Notes*, ACM Press, vol. 20, no. 2, April, 1995, pp. 17–18.
- [Dav95b] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.
- [Day99] Dayani-Fard, H., et al., "Legacy Software Systems: Issues, Progress, and Challenges," IBM Technical Report: TR-74.165-k, April 1999, available at www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html.
- [Dem86] Deming, W. E., *Out of the Crisis*, MIT Press, 1986.
- [DeM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.
- [DeM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995.
- [DeM95a] DeMarco, T., "Lean and Mean," *IEEE Software*, November 1995, pp. 101–102.
- [DeM98] DeMarco, T., and T. Lister, *Peopleware*, 2d ed., Dorset House, 1998.
- [DeM02] DeMarco, T., and B. Boehm, "The Agile Methods Fray," *IEEE Computer*, vol. 35, no. 6, June 2002, pp. 90–92.
- [Den73] Dennis, J., "Modularity," in *Advanced Course on Software Engineering* (F. L. Bauer, ed.), Springer-Verlag, 1973, pp. 128–182.
- [Dev01] Devedzik, V., "Software Patterns," in *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 2001.
- [Dha95] Dhama, H., "Quantitative Metrics for Cohesion and Coupling in Software," *Journal of Systems and Software*, vol. 29, no. 4, April 1995.
- [Dij65] Dijkstra, E., "Programming Considered as a Human Activity," in *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965.
- [Dij72] Dijkstra, E., "The Humble Programmer," 1972 ACM Turing Award Lecture, *CACM*, vol. 15, no. 10, October 1972, pp. 859–866.

- [Dij76a] Dijkstra, E., "Structured Programming," in *Software Engineering, Concepts and Techniques*, (J. Buxton et al., eds.), Van Nostrand-Reinhold, 1976.
- [Dij76b] Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.
- [Dij82] Dijkstra, E., "On the Role of Scientific Thought," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.
- [Dix99] Dix, A., "Design of User Interfaces for the Web," *Proc. User Interfaces to Data Systems Conference*, September 1999, downloadable from www.comp.lancs.ac.uk/computing/users/dixa/topics/webarch/.
- [Dob04] Dobb, F., *ISO 9001:2000 Quality Registration Step-by-Step*, 3d ed., ButterworthHeinemann, 2004.
- [Don99] Donahue, G., S. Weinschenck, and J. Nowicki, "Usability Is Good Business," Compuware Corp., July 1999, available from www.compuware.com.
- [Dre99] Dreilinger, S., "CVS Version Control for Web Site Projects," 1999, available at www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.
- [Dru75] Drucker, P., *Management*, W. H. Heinemann, 1975.
- [Duc01] Ducatel, K., et al., *Scenarios for Ambient Intelligence in 2010*, ISTAG-European Commission, 2001, downloadable from <ftp://ftp.cordis.europa.eu/pub/ist/docs/istagscenarios2010.pdf>.
- [Dun82] Dunn, R., and R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
- [Dun01] Dunaway, D., and S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1,2 Method Description)*, Software Engineering Institute, 2001, downloadable from www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.
- [Dun02] Dunn, W., *Practical Design of Safety-Critical Computer Systems*, William Dunn, 2002.
- [Duy02] VanDuyne, D., J. Landay, and J. Hong, *The Design of Sites*, Addison-Wesley, 2002.
- [Dye92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.
- [Edg95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends*, vol. 2, no. 5, May 1995, pp. 37–42.
- [Eji91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.
- [Elr01] Elrad, T., R. Filman, and A. Bader (eds.), "Aspect Oriented Programming," *Comm. ACM*, vol. 44, no. 10, October 2001, special issue.
- [Eri05] Ericson, C., *Hazard Analysis Techniques for System Safety*, Wiley-Interscience, 2005.
- [Eri08] Erickson, T., *The Interaction Design Patterns Page*, May 2008, available at www.visi.com/~snowfall/InteractionPatterns.html.
- [Eva04] Evans, E., *Domain Driven Design*, Addison-Wesley, 2003.
- [Fag86] Fagan, M., "Advances in Software Inspections," *IEEE Trans. Software Engineering*, vol. 12, no. 6, July 1986.
- [Fel89] Felican, L., and G. Zalateu, "Validating Halstead's Theory for Pascal Programs," *IEEE Trans. Software Engineering*, vol. SE-15, no. 2, December 1989, pp. 1630–1632.
- [Fel07] Feller, J., et al. (eds.), *Perspectives on Free and Open Source Software*, The MIT Press, 2007.
- [Fen91] Fenton, N., *Software Metrics*, Chapman and Hall, 1991.
- [Fen94] Fenton, N., "Software Measurement: A Necessary Scientific Basis," *IEEE Trans. Software Engineering*, vol. SE-20, no. 3, March 1994, pp. 199–206.
- [Fer97] Ferguson, P., et al., "Results of Applying the Personal Software Process," *IEEE Computer*, vol. 30, no. 5, May 1997, pp. 24–31.
- [Fer98] Ferdinandi, P. L., "Facilitating Communication," *IEEE Software*, September 1998, pp. 92–96.

- [Fer00] Fernandez, E. B., and X. Yuan, "Semantic Analysis Patterns," *Proceedings of the 19th Int. Conf. on Conceptual Modeling, ER2000*, Lecture Notes in Computer Science 1920, Springer, 2000, pp. 183–195. Also available from www.cse.fau.edu/~ed/SAPpaper2.pdf.
- [Fir93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993.
- [Fis06] Fisher, R., and D. Shapiro, *Beyond Reason: Using Emotions as You Negotiate*, Penguin, 2006.
- [Fit54] Fitts, P., "The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement," *Journal of Experimental Psychology*, vol. 47, 1954, pp. 381–391.
- [Fle98] Fleming, Q. W., and J. M. Koppelman, "Earned Value Project Management," *CrossTalk*, vol. 11, no. 7, July 1998, p. 19.
- [Fos06] Foster, E., "Quality Culprits," InfoWorld Grip Line Weblog, May 2, 2006, available at http://weblog.infoworld.com/gripline/2006/05/02_a395.html.
- [Fow97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
- [Fow00] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
- [Fow01] Fowler, M., and J. Highsmith, "The Agile Manifesto," *Software Development Magazine*, August 2001, www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm.
- [Fow02] Fowler, M., "The New Methodology," June 2002, www.martinfowler.com/articles/newMethodology.html#N8B.
- [Fow03] Fowler, M., et al., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
- [Fow04] Fowler, M., *UML Distilled*, 3d ed., Addison-Wesley, 2004.
- [Fra93] Frankl, P. G., and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Trans. Software Engineering*, vol. SE-19, no. 8, August 1993, pp. 770–787.
- [Fra03] Francois, A., "Software Architecture for Immersipresence," IMSC Technical Report IMSC-03-001, University of Southern California, December 2003, available at <http://iris.usc.edu/~afrancoi/pdf/sai-tr.pdf>.
- [Fre80] Freeman, P., "The Context of Design," in *Software Design Techniques*, 3d ed. (P. Freeman and A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2–4.
- [Fre90] Freedman, D. P., and G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3d ed., Dorset House, 1990.
- [Gag04] Gage, D., and J. McCormick, "We Did Nothing Wrong," *Baseline Magazine*, March 4, 2004, available at www.baselinemag.com/article2/0,1397,1544403,00.asp.
- [Gai95] Gaines, B., "Modeling and Forecasting the Information Sciences," Technical Report, University of Calgary, Calgary, Alberta, September 1995.
- [Gam95] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Gar84] Garvin, D., "What Does 'Product Quality' Really Mean?" *Sloan Management Review*, Fall 1984, pp. 25–45.
- [Gar87] Garvin D., "Competing on the Eight Dimensions of Quality," *Harvard Business Review*, November 1987, pp. 101–109. A summary is available at www.acm.org/crossroads/xrds6-4/software.html.
- [Gar95] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.
- [Gar08] GartnerGroup, "Understanding Hype Cycles," 2008, available at www.gartner.com/pages/story.php.id.8795.s.8.jsp.

- [Gau89] Gause, D. C., and G. M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
- [Gey01] Geyer-Schulz, A., and M. Hahsler, "Software Engineering with Analysis Patterns," Technical Report 01/2001, Institut für Informationsverarbeitung und -wirtschaft, Wirtschaftsuniversität Wien, November 2001, downloadable from www.wai.wu-wien.ac.at/~hahsler/research/virlib_working2001/virlib/.
- [Gil88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.
- [Gil95] Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter* (online edition, ttn@soft.com), Software Research, January 1995.
- [Gil06] Gillis, D., "Pattern-Based Design," *tehan + lax blog*, September 14, 2006, available at www.teehanlax.com/blog/?p=96.
- [Gla98] Glass, R., "Defining Quality Intuitively," *IEEE Software*, May 1998, pp. 103–104, 107.
- [Gla00] Gladwell, M., *The Tipping Point*, Back Bay Books, 2002.
- [Gli07] Glinz, M., and R. Wieringa, "Stakeholders in Requirements Engineering," *IEEE Software*, vol. 24, no. 2, March–April 2007, pp. 18–20.
- [Glu94] Gluch, D., "A Construct for Describing Software Development Risks," CMU/SEI-94-TR-14, Software Engineering Institute, 1994.
- [Gna99] Gnaho, C., and F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering," *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, May 1999.
- [Gon04] Gonzales, R., "Requirements Engineering," Sandia National Laboratories, a slide presentation, available at www.incose.org/enchantment/docs/04AprRequirementsEngineering.pdf.
- [Gor02] Gordon, B., and M. Gordon, *The Complete Guide to Digital Graphic Design*, Watson-Guptill, 2002.
- [Gor06] Gorton, I., *Essential Software Architecture*, Springer, 2006.
- [Gra87] Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.
- [Gra92] Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.
- [Gra99] Grable, R., et al., "Metrics for Small Projects: Experiences at SED," *IEEE Software*, March 1999, pp. 21–29.
- [Gra03] Gradecki, J., and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.
- [Gru02] Grundy, J., "Aspect-Oriented Component Engineering," 2002, www.cs.auckland.ac.nz/~john-g/aspects.html.
- [Gus89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment," *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, October 1989, pp. 114–117.
- [Gut93] Guttag, J. V., and J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
- [Hac98] Hackos, J., and J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.
- [Hai02] Hailpern, B., and P. Santhanam, "Software Debugging, Testing and Verification," *IBM Systems Journal*, vol. 41, no. 1, 2002, available at www.research.ibm.com/journal/sj/411/hailpern.html.
- [Hal77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.
- [Hal90] Hall, A., "Seven Myths of Formal Methods," *IEEE Software*, September 1990, pp. 11–20.
- [Hal98] Hall, E. M., *Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998.

- [Ham90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate," *Harvard Business Review*, July–August 1990, pp. 104–112.
- [Han95] Hanna, M., "Farewell to Waterfalls," *Software Magazine*, May 1995, pp. 38–46.
- [Har98a] Harmon, P., "Navigating the Distributed Components Landscape," *Cutter IT Journal*, vol. 11, no. 2, December 1998, pp. 4–11.
- [Har98b] Harrison, R., S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Engineering*, vol. SE-24, no. 6, June 1998, pp. 491–496.
- [Her00] Herrmann, D., *Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000.
- [Het84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.
- [Het93] Hetzel, W., *Making Software Measurement Work*, QED Publishing, 1993.
- [Hev93] Hevner, A. R., and H. D. Mills, "Box Structure Methods for System Development with Objects," *IBM Systems Journal*, vol. 31, no. 2, February 1993, pp. 232–251.
- [Hig95] Higuera, R. P., "Team Risk Management," *CrossTalk*, U.S. Dept. of Defense, January 1995, pp. 2–4.
- [Hig00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems*, Dorset House Publishing, 2000.
- [Hig01] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 1," *Cutter IT Journal*, vol. 14, no. 12, December 2001.
- [Hig02a] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 2," *Cutter IT Journal*, vol. 15, no. 1, January 2002.
- [Hig02b] Highsmith, J., *Agile Software Development Ecosystems*, Addison-Wesley, 2002.
- [Hil05] Hildreth, S., "Buggy Software: Up from a Low Quality Quagmire," *Computerworld*, July 25, 2005, available at www.computerworld.com/developmenttopics/development/story/0,10801,103378,00.html.
- [Hil08] Hillside.net, *Patterns Catalog*, 2008, available at <http://hillside.net/patterns/> online patterncatalog.htm.
- [Hob06] Hoberman, S., *Data Modeling Made Simple*, Technics Publications, 2006.
- [Hof00] Hofmeister, C., R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.
- [Hof01] Hofmann, C., et al., "Approaches to Software Architecture," 2001, downloadable from <http://citeseer.nj.nec.com/84015.html>.
- [Hol06] Holzner, S., *Design Patterns for Dummies*, For Dummies Publishers, 2006.
- [Hoo96] Hooker, D., "Seven Principles of Software Development," September 1996, available at <http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment>.
- [Hop90] Hopper, M. D., "Rattling SABRE, New Ways to Compete on Information," *Harvard Business Review*, May–June 1990.
- [Hor03] Horch, J., *Practical Guide to Software Quality Management*, 2d ed., Artech House, 2003.
- [HPR02] Hypermedia Design Patterns Repository, 2002, available at www.designpattern.lu.unisi.ch/index.htm.
- [Hum95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.
- [Hum96] Humphrey, W., "Using a Defined and Measured Personal Software Process," *IEEE Software*, vol. 13, no. 3, May–June 1996, pp. 77–88.
- [Hum97] Humphrey, W., *Introduction to the Personal Software Process*, Addison-Wesley, 1997.
- [Hum98] Humphrey, W., "The Three Dimensions of Process Improvement, Part III: The Team Process," *CrossTalk*, April 1998, available at www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.asp.
- [Hum00] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, 2000.

- [Hun99] Hunt, A., D. Thomas, and W. Cunningham, *The Pragmatic Programmer*, Addison-Wesley, 1999.
- [Hur83] Hurley, R. B., *Decision Tables in Software Engineering*, Van Nostrand-Reinhold, 1983.
- [Hya96] Hyatt, L., and L. Rosenberg, "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality," NASA SATC, 1996, available at http://satc.gsfc.nasa.gov/support/STC_APR96/qualtiy/stc_qual.html.
- [IBM81] "Implementing Software Inspections," course notes, IBM Systems Sciences Institute, IBM Corporation, 1981.
- [IBM03] IBM, *Web Services Globalization Model*, 2003, available at www.ibm.com/develo perworks/webservices/library/ws-global/.
- [IEE93a] *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.
- [IEE93b] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1993.
- [IEE00] IEEE Standard Association, IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000, available at http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html.
- [IFP01] *Function Point Counting Practices Manual*, Release 4.1.1, International Function Point Users Group, 2001, available from www.ifpug.org/publications/manual.htm.
- [IFP05] Function Point Bibliography/Reference Library, International Function Point Users Group, 2005, available from www.ifpug.org/about/bibliography.htm.
- [ISI08] iSixSigma, LLC, "New to Six Sigma: A Guide for Both Novice and Experiences Quality Practitioners," 2008, available at www.isixsigma.com/library/content/six-sigma-newbie.asp.
- [ISO00] ISO 9001: 2000 Document Set, International Organization for Standards, 2000, www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.
- [ISO02] *Z Formal Specification Notation—Syntax, Type System and Semantics*, ISO/IEC 13568:2002, Intl. Standards Organization, 2002.
- [ISO08] ISO SPICE, 2008, www.isospice.com/categories/SPICE-Project/.
- [Ivo01] Ivory, M., R. Sinha, and M. Hearst, "Empirically Validated Web Page Design Metrics," *ACM SIGCHI'01*, March 31–April 4, 2001, available at <http://webtango.berkeley.edu/papers/chi2001/>.
- [Jac75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.
- [Jac92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
- [Jac98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software*, July 1998, pp. 43–45.
- [Jac99] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
- [Jac02a] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More," *Cutter IT Journal*, vol. 15, no. 1, January 2002, pp. 18–24.
- [Jac02b] Jacyntho, D., D. Schwabe, and G. Rossi, "An Architecture for Structuring Complex Web Applications," 2002, available at www2002.org/CDROM/alternate/478/.
- [Jac04] Jacobson, I., and P. Ng, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
- [Jal04] Jalote, P., et al., "Timeboxing: A Process Model for Iterative Software Development," *Journal of Systems and Software*, vol. 70, issue 2, 2004, pp. 117–127. Available at www.cse.iitk.ac.in/users/jalote/papers/Timeboxing.pdf.
- [Jay94] Jaychandra, Y., *Re-engineering the Networked Enterprise*, McGraw-Hill, 1994.
- [Jec06] Jech, T., *Set Theory*, 3d ed., Springer, 2006.
- [Jon86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.
- [Jon91] Jones, C., *Systematic Software Development Using VDM*, 2d ed., Prentice Hall, 1991.

- [Jon96] Jones, C., "How Software Estimation Tools Work," *American Programmer*, vol. 9, no. 7, July 1996, pp. 19–27.
- [Jon98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998.
- [Jon04] Jones, C., "Software Project Management Practices: Failure Versus Success," *CrossTalk*, October 2004. Available at www.stsc.hill.af.mil/crossTalk/2004/10/0410Jones.html.
- [Joy00] Joy, B., "The Future Doesn't Need Us," *Wired*, vol. 8, no. 4, April 2000.
- [Kai02] Kaiser, J., "Elements of Effective Web Design," About, Inc., 2002, available at <http://webdesign.about.com/library/weekly/aa091998.htm>.
- [Kal03] Kalman, S., *Web Security Field Guide*, Cisco Press, 2003.
- [Kan93] Kaner, C., J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2d ed., Van Nostrand-Reinhold, 1993.
- [Kan95] Kaner, C., "Lawyers, Lawsuits, and Quality Related Costs, 1995, available at www.badsoftware.com/plaintif.htm.
- [Kan01] Kaner, C., "Pattern: Scenario Testing" (draft), 2001, available at www.testing.com/testpatterns/patterns/pattern-scenario-testing-kaner.html.
- [Kar94] Karten, N., *Managing Expectations*, Dorset House, 1994.
- [Kau95] Kauffman, S., *At Home in the Universe*, Oxford, 1995.
- [Kaz98] Kazman, R., et al., *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, July 1998.
- [Kaz03] Kazman, R., and A. Eden, "Defining the Terms Architecture, Design, and Implementation," *news@sei interactive*, Software Engineering Institute, vol. 6, no. 1, 2003, available at www.sei.cmu.edu/news-at-sei/columns/the_architect/2003/1q03/architect-1q03.htm.
- [Kei98] Keil, M., et al., "A Framework for Identifying Software Project Risks," *CACM*, vol. 41, no. 11, November 1998, pp. 76–83.
- [Kel00] Kelly, D., and R. Oshana, "Improving Software Quality Using Statistical Techniques, Information and Software Technology," Elsevier, vol. 42, August 2000, pp. 801–807, available at www.eng.auburn.edu/~kchang/comp6710/readings/Improving_Quality_with_Statistical_Testing_InfoSoftTech_August2000.pdf.
- [Ker78] Kernighan, B., and P. Plauger, *The Elements of Programming Style*, 2d ed., McGraw-Hill, 1978.
- [Ker05] Kerievsky, J., *Industrial XP: Making XP Work in Large Organizations*, Cutter Consortium, Executive Report, vol. 6., no. 2, 2005, available at www.cutter.com/content-and-analysis/resource-centers/agile-project-management/sample-our-research/apmr0502.html.
- [Kim04] Kim, E., "A Manifesto for Collaborative Tools," *Dr. Dobb's Journal*, May 2004, available at www.blueoxen.com/papers/0000D/.
- [Kir94] Kirani, S., and W. T. Tsai, "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994.
- [Kiz05] Kizza, J., *Computer Network Security*, Springer, 2005.
- [Knu98] Knuth, D., *The Art of Computer Programming*, three volumes, Addison-Wesley, 1998.
- [Kon02] Konrad, S., and B. Cheng, "Requirements Patterns for Embedded Systems," *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, IEEE, September 2002, pp. 127–136, downloadable from <http://citeseer.ist.psu.edu/669258.html>.
- [Kra88] Krasner, G., and S. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, August–September 1988, pp. 26–49.
- [Kra95] Kraul, R., and L. Streeter, "Coordination in Software Development," *CACM*, vol. 38, no. 3, March 1995, pp. 69–81.

- [Kru05] Krutchen, P., "Software Design in a Postmodern Era," *IEEE Software*, vol. 22, no. 2, March–April, 2005, pp. 16–18.
- [Kru06] Krutchen, P., H. Obbink, and J. Stafford (eds.), "Software Architectural" (special issue), *IEEE Software*, vol. 23, no. 2, March–April, 2006.
- [Kur05] Kurzweil, R., *The Singularity Is Near*, Penguin Books, 2005.
- [Kyb84] Kyburg, H. E., *Theory and Measurement*, Cambridge University Press, 1984.
- [Laa00] Laakso, S., et al., "Improved Scroll Bars," *CHI 2000 Conf. Proc.*, ACM, 2000, pp. 97–98, available at www.cs.helsinki.fi/u/salaakso/patterns/.
- [Lai02] Laitenberger, A., "A Survey of Software Inspection Technologies," in *Handbook on Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 2002.
- [Lam01] Lam, W., "Testing E-Commerce Systems: A Practical Guide," *IEEE IT Pro*, March–April 2001, pp. 19–28.
- [Lan01] Lange, M., "It's Testing Time! Patterns for Testing Software, June 2001, downloadable from www.testing.com/test-patterns/patterns/index.html.
- [Lan02] Land, R., "A Brief Survey of Software Architecture," Technical Report, Dept. of Computer Engineering, Mälardalen University, Sweden, February 2002.
- [Leh97a] Lehman, M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.
- [Leh97b] Lehman, M., et al., "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, downloadable from www.ece.utexas.edu/~perry/work/papers/feast1.pdf.
- [Let01] Lethbridge, T., and R. Laganieri, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.
- [Let03a] Lethbridge, T., Personal communication on domain analysis, May 2003.
- [Let03b] Lethbridge, T., Personal communication on software metrics, June 2003.
- [Lev95] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
- [Lev01] Levinson, M., "Let's Stop Wasting \$78 billion a Year," *CIO Magazine*, October 15, 2001, available at www.cio.com/archive/101501/wasting.html.
- [Lew06] Lewicki, R., B. Barry, and D. Saunders, *Essentials of Negotiation*, McGraw-Hill, 2006.
- [Lie03] Lieberherr, K., "Demeter: Aspect-Oriented Programming," May 2003, available at www.ccs.neu.edu/home/lieber/LoD.html.
- [Lin79] Linger, R., H. Mills, and B. Witt, *Structured Programming*, Addison-Wesley, 1979.
- [Lin88] Linger, R. M., and H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proc. COMPSAC '88*, Chicago, October 1988.
- [Lin94] Linger, R., "Cleanroom Process Model," *IEEE Software*, vol. 11, no. 2, March 1994, pp. 50–58.
- [Lis88] Liskov, B., "Data Abstraction and Hierarchy," *SIGPLAN Notices*, vol. 23, no. 5, May 1988.
- [Liu98] Liu, K., et al., "Report on the First SEBPC Workshop on Legacy Systems," Durham University, February 1998, available at www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html.
- [Lon02] Longstreet, D., "Fundamental of Function Point Analysis," Longstreet Consulting, Inc., 2002, available at www.ifpug.com/fpafund.htm.
- [Lor94] Lorenz, M., and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.
- [Maa07] Maassen, O., and S. Stelling, "Creational Patterns: Creating Objects in an OO System," 2007, available at www.informit.com/articles/article.aspx?p=26452&rl=1.
- [Man81] Mantai, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM*, vol. 24, no. 3, March 1981, pp. 106–113.
- [Man97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.
- [Mar94] Marick, B., *The Craft of Software Testing*, Prentice Hall, 1994.

- [Mar00] Martin, R., "Design Principles and Design Patterns," downloadable from [www .object mentor.com](http://www.objectmentor.com), 2000.
- [Mar01] Marciniak, J. J. (ed.), *Encyclopedia of Software Engineering*, 2d ed., Wiley, 2001.
- [Mar02] Marick, B., "Software Testing Patterns," 2002, [www.testing.com/test-patterns/ index. html](http://www.testing.com/test-patterns/index.html).
- [McC76] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering*, vol. SE-2, December 1976, pp. 308–320.
- [McC77] McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, November 1977.
- [McC94] McCabe, T. J., and A. H. Watson, "Software Complexity," *CrossTalk*, vol. 7, no. 12, December 1994, pp. 5–9.
- [McC96] McConnell, S., "Best Practices: Daily Build and Smoke Test", *IEEE Software*, vol. 13, no. 4, July 1996, pp. 143–144.
- [McC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.
- [McC99] McConnell, S., "Software Engineering Principles," *IEEE Software*, vol. 16, no. 2, March–April 1999, available at www.stevemcconnell.com/ieeesoftware/eic04.htm.
- [McC04] McConnell, S., *Code Complete*, Microsoft Press, 2004.
- [McC05] McCrory, A., "Ten Technologies to Watch in 2006," *SeachCIO.com*, October 27, 2005, available at [http://searchcio.techtarget.com/originalContent/0,289142,sid19_gci1137889,00 .html](http://searchcio.techtarget.com/originalContent/0,289142,sid19_gci1137889,00.html).
- [McDE93] McDermid, J., and P. Rook, "Software Development Process Models," in *Software Engineer's Reference Book*, CRC Press, 1993, pp. 15/26–15/28.
- [McG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes*, vol. 16, no. 4, October 1991, pp. 53–54.
- [McG94] McGregor, J. D., and T. D. Korson, "Integrated Object-Oriented Testing and Development Processes," *Communications of the ACM*, vol. 37, no. 9, September, 1994, pp. 59–77.
- [Men01] Mendes, E., N. Mosley, and S. Counsell, "Estimating Design and Authoring Effort," *IEEE Multimedia*, vol. 8, no. 1, January–March 2001, pp. 50–57.
- [Mer93] Merlo, E., et al., "Reengineering User Interfaces," *IEEE Software*, January 1993, pp. 64–73.
- [Mic08] *Microsoft Accessibility Technology for Everyone*, 2008, available at [www.microsoft.com/ enable/](http://www.microsoft.com/enable/).
- [Mic04] Microsoft, "Prescriptive Architecture: Integration and Patterns," *MSDN*, May 2004, available at <http://msdn2.microsoft.com/en-us/library/ms978700.aspx>.
- [Mic07] Microsoft, "Patterns and Practices," *MSDN*, 2007, available at [http://msdn2.microsoft .com/en-us/library/ms998478.aspx](http://msdn2.microsoft.com/en-us/library/ms998478.aspx).
- [Mil72] Mills, H. D., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, MD, 1972.
- [Mil77] Miller, E., "The Philosophy of Testing," in *Program Testing Techniques*, IEEE Computer Society Press, 1977, pp. 1–3.
- [Mil87] Mills, H. D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, September 1987, pp. 19–25.
- [Mil88] Mills, H. D., "Stepwise Refinement and Verification in Box Structured Systems," *Computer*, vol. 21, no. 6, June 1988, pp. 23–35.
- [Mil00a] Miller, E., "WebSite Testing," 2000, available at [www.soft.com/eValid/Technology/ White. Papers/website.testing.html](http://www.soft.com/eValid/Technology/White.Papers/website.testing.html).
- [Mil00b] Mili, A., and R. Cowan, "Software Engineering Technology Watch," April 6, 2000, available at www.serc.net/projects/TechWatch/NSF%20TechWatch%20Proposal.htm.
- [Min95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.

- [Mon84] Monk, A. (ed.), *Fundamentals of Human-Computer Interaction*, Academic Press, 1984.
- [Mor81] Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3–50.
- [Mor05] Morales, A., "The Dream Team," *Dr. Dobbs Portal*, March 3, 2005, available at www.ddj.com/dept/global/184415303.
- [Mus87] Musa, J. D., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.
- [Mus93] Musa, J., "Operational Profiles in Software Reliability Engineering," *IEEE Software*, March 1993, pp. 14–32.
- [Mut03] Mutafelija, B., and H. Stromberg, *Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech, 2003.
- [Mye78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.
- [Mye79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
- [NAS07] NASA, *Software Risk Checklist*, Form LeR-F0510.051, March 2007, downloadable from <http://osat-ext.grc.nasa.gov/rmo/spa/SoftwareRiskChecklist.doc>.
- [Nau69] Naur, P., and B. Randall (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.
- [Ngu00] Nguyen, H., "Testing Web-Based Applications," *Software Testing and Quality Engineering*, May–June 2000, available at www.stqemagazine.com.
- [Ngu01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.
- [Ngu06] Nguyen, T., "Model-Based Version and Configuration Management for a Web Engineering Lifecycle," *Proc. 15th Intl. World Wide Web Conf.*, Edinburg, Scotland, 2006, download from www2006.org/programme/item.php?id=4552.
- [Nie92] Nierstrasz, O., S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *CACM*, vol. 35, no. 9, September 1992, pp. 160–165.
- [Nie94] Nielsen, J., and J. Levy, "Measuring Usability: Preference vs. Performance," *CACM*, vol. 37, no. 4, April 1994, pp. 65–75.
- [Nie96] Nielsen, J., and A. Wagner, "User Interface Design for the WWW," *Proc. CHI '96 Conf. on Human Factors in Computing Systems*, ACM Press, 1996, pp. 330–331.
- [Nie00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.
- [Nog00] Nogueira, J., C. Jones, and Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering," Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, June 2000, downloadable from www.dodccrp.org/2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.
- [Nor70] Norden, P., "Useful Tools for Project Management" in *Management of Production*, M. K. Starr (ed.), Penguin Books, 1970.
- [Nor86] Norman, D. A., "Cognitive Engineering," in *User Centered Systems Design*, Lawrence Earlbaum Associates, 1986.
- [Nor88] Norman, D., *The Design of Everyday Things*, Doubleday, 1988.
- [Nov04] Novotny, O., "Next Generation Tools for Object-Oriented Development," *The Architecture Journal*, January 2005, available at <http://msdn2.microsoft.com/en-us/library/aa480062.aspx>.
- [Noy02] Noyes, B., "Rugby, Anyone?" *Managing Development* (an online publication of Fawcette Technical Publications), June 2002, www.fawcette.com/resources/managingdev/methodologies/scrum/.
- [Off02] Offutt, J., "Quality Attributes of Web Software Applications," *IEEE Software*, March–April 2002, pp. 25–32.
- [Ols99] Olsina, L., et al., "Specifying Quality Characteristics and Attributes for Web Sites," *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, May 1999. [Ols06] Olsen, G., "From

- COM to Common," *Component Technologies*, ACM, vol. 4, no. 5, June 2006, available at <http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=394>.
- [OMG03a] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.5, March 2003, available from www.rational.com/uml/resources/documentation/.
- [OMG03b] "Object Constraint Language Specification," in *Unified Modeling Language*, v2.0, Object Management Group, September 2003, downloadable from www.omg.org.
- [Orf99] Orfali, R., D. Harkey, and J. Edwards, *Client/Server Survival Guide*, 3d ed., Wiley, 1999.
- [Osb90] Osborne, W. M., and E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle," *IEEE Software*, January 1990, pp. 10–11.
- [OSO08] OpenSource.org, 2008, available at www.opensource.org/.
- [Pag85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.
- [Pal02] Palmer, S., and J. Felsing, *A Practical Guide to Feature Driven Development*, Prentice Hall, 2002.
- [Par72] Parnas, D. L., "On Criteria to Be Used in Decomposing Systems into Modules," *CACM*, vol. 14, no. 1, April 1972, pp. 221–227.
- [Par96a] Pardee, W., *To Satisfy and Delight Your Customer*, Dorset House, 1996.
- [Par96b] Park, R. E., W. B. Goethert, and W. A. Florac, *Goal Driven Software Measurement—A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.
- [Pat07] Patton, J., "Understanding User Centricity," *IEEE Software*, vol. 24, no. 6, November–December, 2007, pp. 9–11.
- [Pau94] Paulish, D., and A. Carleton, "Case Studies of Software Process Improvement Measurement," *Computer*, vol. 27, no. 9, September 1994, pp. 50–57.
- [PCM03] "Technologies to Watch," *PC Magazine*, July 2003, available at www.pcmag.com/article2/0,4149,1130591,00.asp.
- [Per74] Persig, R., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, 1974.
- [Pet06] Pethokoukis, J., "Small Biz Watch: Future Business Trends," *U.S. News & World Report*, January 20, 2006, available at www.usnews.com/usnews/biztech/articles/060120/20sbw.htm.
- [Pha89] Phadke, M. S., *Quality Engineering Using Robust Design*, Prentice Hall, 1989.
- [Pha97] Phadke, M. S., "Planning Efficient Software Tests," *CrossTalk*, vol. 10, no. 10, October 1997, pp. 11–15.
- [Phi98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.
- [Phi02] Phillips, M., "CMMI V1.1 Tutorial," April 2002, available at www.sei.cmu.edu/cmmi/.
- [Pol45] Polya, G., *How to Solve It*, Princeton University Press, 1945.
- [Poo88] Poore, J. H., and H. D. Mills, "Bringing Software Under Statistical Quality Control," *Quality Progress*, November 1988, pp. 52–55.
- [Poo93] Poore, J. H., H. D. Mills, and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software*, vol. 10, no. 1, January 1993, pp. 88–99.
- [Pop03] Poppendieck, M., and T. Poppendieck, *Lean Software Development*, Addison-Wesley, 2003.
- [Pop06a] Poppendieck, LLC, *Lean Software Development*, available at www.poppendieck.com/.
- [Pop06b] Poppendieck, M., and T. Poppendieck, *Implementing Lean Software Development*, Addison-Wesley, 2006.
- [Pop08] Popcorn, F., *Faith Popcorn's Brain Reserve*, 2008, available at www.faithpopcorn.com/.
- [Pot04] Potter, M., *Set Theory and Its Philosophy: A Critical Introduction*, Oxford University Press, 2004.

- [Pow98] Powell, T., *Web Site Engineering*, Prentice Hall, 1998.
- [Pow02] Powell, T., *Web Design*, 2d ed., McGraw-Hill/Osborne, 2002.
- [Pre94] Premerlani, W., and M. Blaha, "An Approach for Reverse Engineering of Relational Databases," *CACM*, vol. 37, no. 5, May 1994, pp. 42–49.
- [Pre88] Pressman, R., *Making Software Engineering Happen*, Prentice Hall, 1988.
- [Pre05] Pressman, R., *Adaptable Process Model*, revision 2.0, R. S. Pressman & Associates, 2005, available at www.rspa.com/apm/index.html.
- [Pre08] Pressman, R., and D. Lowe, *Web Engineering: A Practitioner's Approach*, McGraw-Hill, 2008.
- [Put78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem," *IEEE Trans. Software Engineering*, vol. SE-4, no. 4, July 1978, pp. 345–361.
- [Put92] Putnam, L., and W. Myers, *Measures for Excellence*, Yourdon Press, 1992.
- [Put97a] Putnam, L., and W. Myers, "How Solved Is the Cost Estimation Problem?" *IEEE Software*, November 1997, pp. 105–107.
- [Put97b] Putnam, L., and W. Myers, *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, 1997.
- [Pyz03] Pyzdek, T., *The Six Sigma Handbook*, McGraw-Hill, 2003.
- [QAI08] *A Software Engineering Curriculum*, QAI, 2008, information can be obtained at www.qaieschool.com/innerpages/offer.asp.
- [QSM02] "QSM Function Point Language Gearing Factors," Version 2.0, Quantitative Software Management, 2002, www.qsm.com/FPGearing.html.
- [Rad02] Radice, R., *High-Quality Low Cost Software Inspections*, Paradoxicon Publishing, 2002.
- [Rai06] Raiffa, H., *The Art and Science of Negotiation*, Belknap Press, 2005.
- [Ree99] Reel, J. S., "Critical Success Factors in Software Projects," *IEEE Software*, May 1999, pp. 18–23.
- [Ric01] Ricadel, A., "The State of Software Quality," *InformationWeek*, May 21, 2001, available at www.informationweek.com/838/quality.htm.
- [Ric04] Rico, D., *ROI of Software Process Improvement*, J. Ross Publishing, 2004. A summary article can be found at <http://davidfrico.com/rico03a.pdf>.
- [Roc94] Roche, J. M., "Software Metrics and Measurement Principles," *Software Engineering Notes*, ACM, vol. 19, no. 1, January 1994, pp. 76–85.
- [Roc06] *Graphic Design That Works*, Rockport Publishers, 2006.
- [Roe00] Roetzheim, W., "Estimating Internet Development," *Software Development*, August 2000, available at www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm.
- [Roo96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes," 1996, available at www.imd.ch/fac/roos/paper_po.html.
- [Ros75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer*, vol. 8, no. 5, May 1975.
- [Ros04] Rosenhainer, L., "Identifying Crosscutting Concerns in Requirements Specifications," 2004, available at <http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/Papers/Rosenhainer.pdf>.
- [Rou02] Rout, T (project manager), *SPICE: Software Process Assessment—Part 1: Concepts and Introductory Guide*, 2002, downloadable from www.sqi.gu.edu.au/spice/suite/download.html.
- [Roy70] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. WESCON*, August 1970.
- [Roz05] Rozanski, N., and E. Woods, *Software Systems Architecture*, Addison-Wesley, 2005.
- [Rub88] Rubin, T., *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.

- [Rum91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [Sar06] Sarwate, A., "Hot or Not: Web Application Vulnerabilities," *SC Magazine*, December 27, 2006, available at <http://scmagazine.com/us/news/article/623765/hot-not-web-application-vulnerabilities>.
- [Sca00] Scacchi, W., "Understanding Software Process Redesign Using Modeling, Analysis, and Simulation," *Software Process Improvement and Practice*, Wiley, 2000, pp. 185–195, downloadable at www.ics.uci.edu/~wscacchi/Papers/Software_Process_Redesign/SPIP-ProSim99.pdf.
- [Sce02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.
- [Sch95] Schwabe, D., and G. Rossi, "The Object-Oriented Hypermedia Design Model," *CACM*, vol. 38, no. 8, August 1995, pp. 45–46.
- [Sch96] Schorsch, T., "The Capability Im-Maturity Model," *CrossTalk*, November 1996, available at www.stsc.hill.af.mil/crosstalk/1996/11/xt96d11h.asp.
- [Sch98a] Schneider, G., and J. Winters, *Applying Use Cases*, Addison-Wesley, 1998.
- [Sch98b] Schwabe, D., and G. Rossi, "Developing Hypermedia Applications Using OOHDM," *Proc. Workshop on Hypermedia Development Process, Methods and Models, Hypertext '98*, 1998, downloadable from <http://citeseer.nj.nec.com/schwabe98developing.html>.
- [Sch98c] Schulmeyer, G. C., and J. I. McManus (eds.), *Handbook of Software Quality Assurance*, 3d ed., Prentice Hall, 1998.
- [Sch99] Schneidewind, N., "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics," *IEEE Trans. SE*, vol. 25, no. 6, November–December 1999, pp. 768–781, downloadable from www.dacs.dtic.mil/topics/reliability/IEEETrans.pdf.
- [Sch01a] Schwabe, D., G. Rossi, and Barbosa, S., "Systematic Hypermedia Application Design Using OOHDM," 2001, available at www-di.inf.puc-rio.br/~schwabe/HT96WWW/section1.html.
- [Sch01b] Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM*, Prentice Hall, 2001.
- [Sch02] Schwaber, K., "Agile Processes and Self-Organization," Agile Alliance, 2002, www.aanpo.org/articles/index.
- [Sch03] Schlickman, J., *ISO 9001: 2000 Quality Management System Design*, Artech House Publishers, 2003.
- [Sch06] Schmidt, D., "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, February 2006, pp. 25–31.
- [SDS08] Spice Document Suite, "The SPICE and ISO Document Suite," ISO-Spice, 2008, available at www.isospice.com/articles/9/1/SPICE-Project/Page1.html.
- [Sea93] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout," *IEEE Trans. Software Engineering*, vol. SE-19, no. 7, July 1993, pp. 707–719.
- [SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates," 2003, available at <http://seeri.etsu.edu/default.htm>.
- [SEI00] SCAMPI, VI.0 Standard CMMI @Assessment Method for Process Improvement: Method Description, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, downloadable from www.sei.cmu.edu/publications/documents/00.reports/00tr009.html.
- [SEI02] "Maintainability Index Technique for Measuring Program Maintainability," SEI, 2002, available at www.sei.cmu.edu/str/descriptions/mitmpm_body.html.
- [SEI08] "The Ideal Model," Software Engineering Institute, 2008, available at www.sei.cmu.edu/ideal/.
- [Sha95a] Shaw, M., and D. Garlan, "Formulations and Formalisms in Software Architecture," *Volume 1000—Lecture Notes in Computer Science*, Springer-Verlag, 1995.
- [Sha95b] Shaw, M., et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Engineering*, vol. SE-21, no. 4, April 1995, pp. 314–335.

- [Sha96] Shaw, M., and D. Garlan, *Software Architecture*, Prentice Hall, 1996.
- [Sha05] Shalloway, A., and J. Trott, *Design Patterns Explained*, 2d ed., Addison-Wesley, 2005.
- [Shn80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.
- [Shn04] Shneiderman, B., and C. Plaisant, *Designing the User Interface*, 4th ed., Addison-Wesley, 2004.
- [Sho83] Shooman, M. L., *Software Engineering*, McGraw-Hill, 1983.
- [Sim05] Simsion, G., and G. Witt, *Data Modeling Essentials*, 3d ed., Morgan Kaufman, 2005.
- [Sin99] Singpurwalla, N., and S. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk*, Springer-Verlag, 1999.
- [Smi99] Smith, J., "The Estimation of Effort Based on Use Cases," Rational Software Corp., 1999, downloadable from www.rational.com/media/whitepapers/finalTP171.PDF.
- [Smi05] Smith, D., *Reliability, Maintainability and Risk*, 7th ed., Butterworth-Heinemann, 2005.
- [Sne95] Sneed, H., "Planning the Reengineering of Legacy Systems," *IEEE Software*, January 1995, pp. 24–25.
- [Sne03] Snee, R., and R. Hoerl, *Leading Six Sigma*, Prentice Hall, 2003.
- [Sol99] van Solingen, R., and E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.
- [Som97] Somerville, I., and P. Sawyer, *Requirements Engineering*, Wiley, 1997.
- [Som05] Somerville, I., "Integrating Requirements Engineering: A Tutorial," *IEEE Software*, vol. 22, no. 1, January–February 2005, pp. 16–23.
- [SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction," Version 1.0, ISO/IEC JTC1, 1999.
- [Spl01] Splaine, S., and S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001.
- [Spo02] Spolsky, J., "The Law of Leaky Abstractions," November 2002, available at www.joelonsoftware.com/articles/LeakyAbstractions.html.
- [Sri01] Sridhar, M., and N. Mandyam, "Effective Use of Data Models in Building Web Applications," 2001, available at www2002.org/CDROM/alternate/698/.
- [SSO08] Software-Supportability.org, www.software-supportability.org/, 2008.
- [Sta97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice*, Addison-Wesley, 1997.
- [Sta97b] Statz, J., D. Oxley, and P. O'Toole, "Identifying and Managing Risks for Software Process Improvement," *CrossTalk*, April 1997, available at www.stsc.hill.af.mil/crosstalk/1997/04/identifying.asp.
- [Ste74] Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, 1974, pp. 115–139.
- [Ste93] Stewart, T. A., "Reengineering: The Hot New Managing Tool," *Fortune*, August 23, 1993, pp. 41–48.
- [Ste99] Stelzer, D., and W. Mellis, "Success Factors of Organizational Change in Software Process Improvement," *Software Process Improvement and Practice*, vol. 4, no. 4, Wiley, 1999, downloadable from www.systementwicklung.uni-koeln.de/forschung/artikel/dokumente/successfactors.pdf.
- [Ste03] Stephens, M., and D. Rosenberg, *Extreme Programming Refactored*, Apress, 2003.
- [Sto05] Stone, D., et al., *User Interface Design and Evaluation*, Morgan Kaufman, 2005.
- [Tai89] Tai, K. C., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58–61.
- [Tay90] Taylor, D., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.
- [Tha97] Thayer, R. H., and M. Dorfman, *Software Requirements Engineering*, 2d ed., IEEE Computer Society Press, 1997.

- [The01] Thelin, T., H. Petersson, and C. Wohlin, "Sample Driven Inspections," *Proc. of Workshop on Inspection in Software Engineering (WISE'01)*, Paris, France, July 2001, pp. 81–91, downloadable from <http://www.cas.mcmaster.ca/wise/wise01/ThelinPeterssonWohlin.pdf>.
- [Tho92] Thomsett, R., "The Indiana Jones School of Risk Management," *American Programmer*, vol. 5, no. 7, September 1992, pp. 10–18.
- [Tic05] *TickIT*, 2005, www.tickit.org/.
- [Tid02] Tidwell, J., "IU Patterns and Techniques," May 2002, available at <http://time-tripper.com/uipatterns/index.html>.
- [Til93] Tillmann, G., *A Practical Guide to Logical Data Modeling*, McGraw-Hill, 1993.
- [Til00] Tillman, H., "Evaluating Quality on the Net," Babson College, May 30, 2000, available at www.hopetillman.com/findqual.html#2.
- [Tog01] Tognozzi, B., "First Principles," *askTOG*, 2001, available at www.asktog.com/basics/firstPrinciples.html.
- [Tra95] Tracz, W., "Third International Conference on Software Reuse—Summary," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 21–22.
- [Tre03] Trivedi, R, *Professional Web Services Security*, Wrox Press, 2003.
- [Tri05] Tricker, R., and B. Sherring-Lucas, *ISO 9001: 2000 In Brief*, 2d ed., ButterworthHeinemann, 2005.
- [Tyr05] Tyree, J., and A. Akerman, "Architectural Decisions: Demystifying Architecture," *IEEE Software*, vol. 22, no. 2, March–April, 2005.
- [Uem99] Uemura, T., S. Kusumoto, and K. Inoue: "A Function Point Measurement Tool for UML Design Specifications," *Proc. of Sixth International Symposium on Software Metrics*, IEEE, November 1999, pp. 62–69.
- [Ull97] Ullman, E., *Close to the Machine: Technophilia and Its Discontents*, City Lights Books, 2002.
- [UML03] The UML Café, "Customers Don't Print Themselves," May 2003, available at www.theumlcafe.com/a0079.htm.
- [Uni03] Unicode, Inc., *The Unicode Home Page*, 2003, available at www.unicode.org/.
- [USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), January 20, 1987.
- [Vac06] Vacca, J., *Practical Internet Security*, Springer, 2006.
- [Van89] Van Vleck, T., "Three Questions About Each Bug You Find," *ACM Software Engineering Notes*, vol. 14, no. 5, July 1989, pp. 62–63.
- [Van02] Van Steen, M., and A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.
- [Ven03] Venners, B., "Design by Contract: A Conversation with Bertrand Meyer," *Artima Developer*, December 8, 2003, available at www.artima.com/intv/contracts.html.
- [Wal03] Wallace, D., I. Raggett, and J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.
- [War74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.
- [War07] Ward, M., "Using VoIP Software Building zBlocks—A Look at the Choices," TMNNet, 2007, available at www.tmcnet.com/voip/0605/featurearticle-using-voip-software-building-blocks.htm.
- [Web05] Weber, S., *The Success of Open Source*, Harvard University Press, 2005.
- [Wei86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.
- [Wel99] Wells, D., "XP—Unit Tests," 1999, available at www.extremeprogramming.org/rules/unittests.html.
- [Wel01] vanWelie, M., "Interaction Design Patterns," 2001, available at www.welie.com/patterns/.

- [Whi95] Whittle, B., "Models and Languages for Component Description and Reuse," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 76–89.
- [Whi97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.
- [Wie02] Wiegers, K., *Peer Reviews in Software*, Addison-Wesley, 2002.
- [Wie03] Wiegers, K., *Software Requirements*, 2d ed., Microsoft Press, 2003.
- [Wil93] Wilde, N., and R. Huitt, "Maintaining Object-Oriented Software," *IEEE Software*, January 1993, pp. 75–80.
- [Wil97] Williams, R. C., J. A. Walker, and A. J. Dorofee, "Putting Risk Management into Practice," *IEEE Software*, May 1997, pp. 75–81.
- [Wil99] Wilkens, T. T., "Earned Value, Clear and Simple," Primavera Systems, April 1, 1999, p. 2.
- [Wil00] Williams, L., and R. Kessler, "All I Really Need to Know about Pair Programming I Learned in Kindergarten," *CACM*, vol. 43, no. 5, May 2000, available at <http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF>.
- [Wil05] Willoughby, M., "Q&A: Quality Software Means More Secure Software," *Computerworld*, March 21, 2005, available at www.computerworld.com/securitytopics/security/story/0,10801,91316,00.html.
- [Win90] Wing, J. M., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, September 1990, pp. 8–24.
- [Wir71] Wirth, N., "Program Development by Stepwise Refinement," *CACM*, vol. 14, no. 4, 1971, pp. 221–227.
- [Wir90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Prentice Hall, 1990.
- [WMT02] Web Mapping Testbed Tutorial., 2002, available at www.webmapping.org/vcgdocuments/vcgTutorial/.
- [Woh94] Wohlin, C., and P. Runeson, "Certification of Software Components," *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 494–499.
- [Wor04] World Bank, *Digital Technology Risk Checklist*, 2004, downloadable from www.moonv6.org/lists/att-0223/WWBANK_Technology_Risk_Checklist_Ver_6point1.pdf.
- [W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, available at www.w3.org/TR/2003/WD-WCAG20-20030624/.
- [Yac03] Yacoub, S., et al., *Pattern-Oriented Analysis and Design*, Addison-Wesley, 2003.
- [You75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, 1975.
- [You79] Yourdon, E., and L. Constantine, *Structured Design*, Prentice Hall, 1979.
- [You95] Yourdon, E., "When Good Enough Is Best," *IEEE Software*, vol. 12, no. 3, May 1995, pp. 79–81.
- [You01] Young, R., *Effective Requirements Practices*, Addison-Wesley, 2001.
- [Zah90] Zahniser, R. A., "Building Software in Groups," *American Programmer*, vol. 3, nos. 7–8, July–August 1990.
- [Zah94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development*, March 1994, pp. 35–38.
- [Zha98] Zhao, J., "On Assessing the Complexity of Software Architectures," *Proc. Intl. Software Architecture Workshop*, ACM, Orlando, FL, 1998, pp. 163–167.
- [Zha02] Zhao, H., "Fitt's Law: Modeling Movement Time in HCI," *Theories in Computer Human Interaction*, University of Maryland, October 2002, available at www.cs.umd.edu/class/fall2002/cmsc838s/tichi/fitts.html.
- [Zul92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers," *American Programmer*, February 1992, pp. 28–41.
- [Zus90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990.
- [Zus97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

A

Abstração 212
 Abstração de dados, 213
 Abstração procedural, 213
 Ação, 40
 Acessibilidade, 305, 478
 Acompanhamento de dependências, 520
 Acoplamento, 216, 267
 categorias, 267
 métricas de, 555
 Agilidade, 82
 custo da mudança, 83
 fatores humanos, 86
 políticas de, 85
 princípios de, 84
 processo, 84
 Agregação, 730
 Agrupamento, 276
 Alternativas arquitetônicas, 239
 Ambiente de trabalho, 300
 Amplificação de defeito, 375
 Análise de domínio, 153
 Análise de imprevistos, 397, 658
 Análise de inventário, 669
 Análise de requisitos, 151
 limites na, 197
 objetivos, 152
 regras práticas, 152
 Análise de usuário, 294
 Análise de Valor Agregado (*Earned Value Analysis* - EVA), 643
 Análise de valor-limite, 442, 480
 Análise estruturada, 154, 182
 Análise sintática, 167, 183
 Aperfeiçoamento do Processo de Software (*Software Process Improvement* - SPI), 574, 682
 abordagens para, 58
 avaliação, 686
 componentes, 683
 definição de, 683
 educação, 687
 elementos de, 686
 estrutura, 683
 fatores críticos de sucesso, 690
 gerenciamento de risco, 689
 instalação/migração, 689
 processo, 686
 ROI, 697
 seleção/justificação, 688
 tendências, 698
 Aplicações Web (WebApp), 35, 37
 características das, 37
 conceitos de teste, 468
 dimensões de qualidade, 469
 erros, 469
 estratégia de teste, 470
 gerenciamento de configuração, 528
 gerenciando alterações, 531
 métricas, 557
 modelagem de navegação, 201
 modelagem de requisitos, 197
 modelo de configuração, 201
 modelo de conteúdo, 199
 modelo de interação, 200
 modelo funcional, 201
 objetos configuração, 529
 padrões de projeto, 333
 pirâmide de projeto, 342
 planejamento de teste, 471
 processo de teste, 471
 projeto da interface de usuário, 306
 projeto de arquitetura, 346
 projeto de interface, 306, 310, 341
 projeto de navegação, 350
 projeto em nível de componente, 274, 352
 projeto estético, 344
 qualidade de projeto, 339
 Arquétipos, 241
 Arquitetura,
 avaliação da, 244
 centralizada em dados, 235
 complexidade da, 246
 definição de, 213, 230
 em camadas, 237
 estilos, 234
 fluxo de dados, 236
 gêneros da, 232
 importância da, 231
 instâncias de, 243
 MVC, 349
 orientada a objetos, 237

 padrões, 327
 refinamento da, 242
 tipos de, 235
 Arquitetura centralizada em dados, 235
 Arquitetura de chamadas e retornos, 236
 Arquitetura de conteúdo, 347
 Arquitetura de fluxo de dados, 236
 Arquitetura de software, *veja também* Arquitetura, 213
 Árvore de dados, 199
 Árvores de decisão, 625
 Aspectos, 70, 217
 Aspectos de intersecção, 217
 Associações, 176
 Atividade, *veja também* atividade de Estrutura, 400
 Atividades de apoio (*Umbrella activities*), 41
 Atores, 138
 Atributos, 746
 especificando, 169
 Atributos de dados, 163
 Auditoria, 534
 Auditoria de configuração, 526
 Auto-organização, 86
 Autoridade de Controle de Alterações (*Change Control Authority* - CCA), 524
 Avaliação de processo, 58
 AVC-EVC, função, *CasaSegura*, 157

B

Biblioteca de reutilização, 284
 Bootstrap, 696
 Bugs, características dos, 421

C

Cadeia definição-uso, 438
 Caminho crítico, 631
 Caminhos de programas independentes, 433
 Campos de aplicação, 34
 Caos, 59
 Carta de estrutura, 261
 Cartões CRC (*CRC cards*), 89
CasaSegura, 47, 65, 66

- abordagem de métricas, 596
- acompanhamento de projeto, 643
- acoplamento, 268
- análise de domínio, 154
- análise de risco, 755
- análise gramatical, 169
- arquetipos, 242
- árvore de dados, 199
- atores, 139
- avaliação de arquitetura, 246
- casos de uso, 140, 156, 160, 161
- cenário preliminar de usuário, 136
- classe de projeto, 220
- coesão, 267
- complexidade ciclomática, 434
- comunicação com o cliente, 114
- conceitos de projeto, 218
- debates de métricas, 543
- depuração, 423
- desenvolvimento ágil, 93
- design vs .codificação, 209
- diagrama de atividades, 202
- diagrama de classe, 171
- diagrama de contexto, 241
- diagrama de estado, 186
- estimativa, 613
- estrutura de equipe, 573
- função ACS-DCV, 157
- iniciação de projeto, 48
- layout da tela, 301
- métricas CK, 553
- métricas de projeto, 587
- modelagem de comportamento, 144
- modelo de fluxo de dados, 182, 187, 545
- modelos CRC, 176
- modelos de classe, 170
- narrativa de produto, 134
- negociação, 146
- NSUs, 350
- OCP em ação, 264
- padrões, 330
- pontos de função, 545
- preparações de teste, 405
- problemas SCM, 526
- projeto de caso de teste, 430
- projeto de conteúdo, 345
- projetoUI, 296
- PSPEC, 187
- refinamento de arquitetura, 238, 254
- regras de ouro, 290
- representação da interface, 224
- reunião de requisitos, 135
- revisão de design de interface, 309
- revisões, 384
- terceirização, 625
- teste de classe, 463
- validação, 418
- WebApp, 197, 479
- Caso de teste, 435
- Casos de uso, 136
 - criando, 155
 - diagrama, 731
 - exceções, 159
 - formal, 159
 - isolando eventos, 195
 - modelos, 161
 - projeto de interface, 295
 - refinando, 158
 - template, 137
- Cenários de uso, 136
- Certificação, 492, 499
- Ciclo de vida clássico, 59
- Ciência do software, 559
- Classe composta de agregação, 174
- Classes,
 - conceitos básicos, 744
 - métodos de teste, 462
 - métricas orientadas a, 551
 - projetos, de, 748
- Cientes, 113
- CMM das pessoas, 695
- COCOMO II, 619
- Coesão, 216, 266
 - métricas de, 555
 - tipos de, 266
- Classes análise,
 - atributos, 170
 - critério de seleção, 167
 - definindo operações, 170
 - diagramas de estado para, 190
 - identificação de, 167
 - tipos de, 172
- Classes de projeto,
 - características das, 219
 - tipos de, 219
- Colaboração, 132, 174
- Colocado (*Check-in*), 524
- Complexidade,
 - arquitetura da, 246
 - métricas orientadas a, 556
- Complexidade ciclomática, 434
- Componentes,
 - adaptação, 305
 - baseado em classes, 262
 - classificando, 283
 - composição, 281
 - convenções de nomes, 266
 - definição de, 258
 - dependências, 266
 - empacotamento, 281
 - interfaces, 265
 - métricas para, 554
 - projeto de, 224
 - qualificação de, 280
 - recuperação de, 284
 - refinamento da arquitetura em, 242
 - visão orientada a objetos, 258
 - visão relacionada com processos, 262
 - visão tradicional, 298, 260
- Composição, 730
- Computação mundial aberta, 35, 706
- Comunicação, 40, 112
- Concepção, 127
- Concretização, 263
- Condição-Transição-Consequência (*Condition-transition-consequence - CTC*), 656
- Confiabilidade, 362
 - definição de, 362
 - métricas, 362
- Confiabilidade do software (*veja* Confiabilidade)
- Conjunto de métricas CK, 551
- Conjunto de métricas MOOD, 553
- Conjunto de modificações, 523
- Conjunto de tarefas, 53, 54

exemplo, 635
 identificando, 55
 Construção, 40, 120
 Continuidade do fluxo de informações, 182
 Controle de acesso, 524
 Controle de mudança, 524
 descrição de processo, 524
 tipos de, 525
 Controle de projeto, 640
 Controle de qualidade de software, 370
 Controle de sincronização, 524
 Controle de versão, 523, 534
 Correção, 594
 verificação, 492, 497
 Cronograma, 629
 conceitos, 630
 princípios, 632
 WebApp, 641
 Crystal, 97
 Curva de defeitos, 33
 Curva Putnam-Norden-Rayleigh (PNR), 633
 Custos,
 defeitos, 374
 mudança, 83
 qualidade, 366

D

Decisão fazer/comprar, 624
 Decisões de arquitetura, 232
 modelo para, 233
 Decomposição de problema, 574
 Defeitos, 374
 Densidade de erro, 377
 Dependências, 177, 729
 Depuração, 421
 automática, 424
 considerações psicológicas, 422
 estratégias, 423
 ferramentas, 424
 processo, 421
 táticas, 423
 Desenvolvimento baseado em componente, 33, 69, 279
 Desenvolvimento baseado em histórico, 91
 Desenvolvimento colaborativo, 712

Desenvolvimento de Software Adaptativo (*Adaptive Software Development - ASD*), 94
 Desenvolvimento de Software Enxuto (*Lean Software Development - LSD*), 99
 Desenvolvimento Dirigido a Funcionalidades (*Feature Driven Development - FDD*), 98
 Desenvolvimento orientado a aspecto, 70
 Desgaste, 32
 Design arquitetônico, *veja também* Design, arquitetônico
 diagrama de contexto, 240
 elementos de, 222
 métricas, 547
 WebApps, 342
 Design de Interface de Usuário (*veja também* Design, interface de usuário), 287
 Design gráfico, WebApps, 344
 Diagrama de atividades, 161, 272, 737
 Diagrama de comunicação, 734
 Diagrama de estado, 144, 273, 739
 Diagrama de fluxo de dados, 182
 criação de, 183
 em nível de contexto, 183
 Diagrama de raias, 162
 Diagrama de sequência, 192, 194, 732
 Diagramas de classe, 727
 Dimensionamento do software, 610
 Disponibilidade, 339
 Disponibilização da Função de Qualidade (*Quality Function Deployment - QFD*), 136
 Distribuição, 41
 diagramas, 730
 princípios, 121
 teste, 420

E

Eficiência, 363

Eficiência de Remoção de Defeitos (*Defect Removal Efficiency - DRE*), 595
 Elaboração, 128, 217, 296
 Elaboração de objeto, 296
 Encapsulamento de informações, 215
 Engenharia concorrente, 67
 Engenharia de domínio, 280
 Engenharia de requisitos, 127, 713
 Engenharia de software baseada em componentes (*Component-based software engineering - CBSE*), 379
 Engenharia de Software Sala Limpa, 69, 492
 estratégia, 492
 modelo de processo, 492
 projeto, 492
 teste estatístico de uso, 498
 Engenharia de software, ambientes, 717
 camadas, 39
 controlada por modelo, 714
 controladas por teste, 715
 definição da, 38
 éticas, 725
 prática, 42, 108
 princípios fundamentais, 109
 princípios gerais, 44
 realidades, 38
 Engenharia de software, aspectos econômicos da, 669
 modelo de processo, 668
 Engenharia direta, 670, 375
 arquiteturas orientadas a objeto, 677
 cliente-servidor, 676
 Engenharia reversa, 669, 670
 dados, 672
 interface de usuário, 673
 processamento, 672
 Entidade-relacionamento (ER)
 diagramas, 165
 Equação de software, 621
 Equipe, 570
 ágil, 572

- auto-organização, 86
 - consistentes, 571
 - estruturas, 570
 - líderes, 568
 - tendências futuras, 709
 - Equipe de software, (*veja* Equipe)
 - Erros
 - correção de, 368, 424
 - definição de, 374
 - estimativa, 605
 - tratamento, 304
 - Escalabilidade, 340
 - Escopo, 574
 - Escopo do software, 606
 - Especificação, 129
 - Especificação da estrutura de caixa, 492, 495
 - Especificação de controle (CSPEC), 185
 - Especificação de processo (PSPEC), 188
 - Especificação de requisitos, template, 129
 - Especificação funcional, 493
 - Estereotipo, 177, 728
 - Estilos, 234
 - Estimativa, 609
 - automática, 618
 - baseada em FP, 613
 - baseada em LOC, 611
 - baseada em problema, 611
 - baseada em processo, 614
 - caso de uso, com, 616
 - desenvolvimento ágil, 622
 - exemplo de, 615
 - modelos empíricos, 618
 - projetos orientados a objeto, 622
 - reconciliando, 617
 - técnicas de decomposição, 610
 - WebApps, 623
 - Estrutura de Divisão de Trabalho (*Work Breakdown Structure - WBS*), 637
 - Estrutura de processo, 40, 44
 - Estruturas, 320, 683
 - Estruturas de arquitetura, 235
 - em rede, 348
 - grade, 347
 - hierárquico, 348
 - linear, 347
 - Ética, 725
 - Eventos, 188
 - Evolução do software, 663
 - Exibir conteúdo, 299
 - Exposição ao risco, 655
- F**
- Facilidade de manutenção, 363, 594, 663
 - Falha, 395
 - Falha ao Longo do Tempo (*Failures-in-time - FIT*), 396
 - Fatoração, 251
 - Fatores humanos, agilidade, 86
 - Feature definition, 86
 - Ferramentas,
 - ADLs, 257
 - análise estruturada, 188
 - BPR, 667
 - CBSE, 284
 - cronograma, 638
 - CVS, 524
 - depuração, 424
 - desenvolvimento ágil, 101
 - desenvolvimento de caso de uso, 142
 - desenvolvimento UI, 305
 - engenharia de requisitos, 130
 - engenharia reversa, 673
 - estimativa, 623
 - gerenciamento de alterações, 532, 533
 - gerenciamento de processo, 71
 - gerenciamento de projeto, 579
 - gerenciamento de risco, 660
 - gerenciamento de teste, 420
 - métodos formais, 510
 - métricas de produto, 562
 - métricas WebApp, 559
 - modelagem de análise com UML, 1+2
 - modelagem de dados, 165
 - modelagem de processo, 77
 - projeto de arquitetura, 244
 - projeto de caso de teste, 444
 - projeto e processo, 593
 - reestruturação, 675
 - SQA, 398
 - suporte SCM, 527
 - tendências, 716
 - teste WebApp, 487
 - Ferramentas de software (*veja* Ferramentas)
 - Fluxo de processo, 43
 - Fluxo de trabalho, 59, 297
 - Folha de informações de risco, 658
 - Funcionalidade, 362
- G**
- Garantia da Qualidade do Software (*Software Quality Assurance - SQA*), 370, 387
 - abordagens formais, 391
 - atributos, 391
 - elementos da, 388
 - estatística, 393
 - métricas, 391
 - objetivos, 391
 - planejamento, 398
 - tarefas, 392
 - Gêneros, 232
 - Gerenciamento de projeto,
 - conceitos, 566
 - práticas vitais, 579
 - Gerenciamento de risco, 648, 689
 - Gestão de alterações, 531
 - Gestão de conteúdo, 530
 - Gestão de requisitos, 130
 - Globalização, 704
 - Gráfico de Gantt, 638
 - Grupo Independente de Teste (*Independent Teste Group - ITG*), 403
 - Guarda, 190
- H**
- Herança, 747
 - Histórias de usuários, 88
- I**
- IA poderosa, 724
 - Incremento de software, 41
 - Independência funcional, 216
 - Indicador, 539
 - Índice de maturidade de software, 562
 - Informação,
 - espectro, 726
 - representação, 726

Informação composta, 163
 Inspeções (*veja* Revisões)
 Integração do Modelo de Maturidade de Capacidade (*Capability Maturity Model Integration - CMMI*), 691, 694
 Integridade, 595
 Inteligência ambiente, 706
 Interessados (*Stakeholders*), 40, 114, 569
 identificação, 131
 múltiplos pontos de vista, 131
 Interface
 análise, 294
 mecanismos, teste de, 475
 projeto, 222, 293
 semânticas, teste de, 477
 teste, 475
 Interface de usuário,
 análise de, 291
 modelo de projeto, 291
 padrões, 330
 teste de, 475
 Internacionalização, 305
 Invariante de dados, 501
 ISO 9001:2000, 58, 397
 ISO 9126, fatores de qualidade, 362
 Itens de Configuração de Software (*Software Configuration Items - SCIs*), 514, 518
 cenário, 515
 elementos da, 516
 padrões, 534
 processo, 521
 repositório, 519
 tarefas, 521
 WebApp, 528

L

Legibilidade, 308
 Lei de Fitt, 307
 Levantamento, 128, 133
 artefatos de requisitos, 137
 Líder de revisão, 381
 Linguagem de Descrição Arquitetônica (*Architectural Description Language - ADL*), 214, 247

Linguagem de especificação Z, 508
 Linguagem de Projeto de Programas (*Program design language - PDL*), 278
 Linguagens de padrões, 321
 Linhas de código (LOC), 590
 Lista de controle de validação de requisitos, 130
 Lista de problemas, 382
 Lista de verificação de itens de risco, 651

M

Manifesto ágil, 81
 Manutenção, 33, 663
 métricas, para, 562
 Manutenção de software, *veja também* Manutenção, 663
 Mapeamento de arquitetura, 247
 Mapeamento de arquitetura, 247
 Matriz de grafos, 436, 440
 Mecanismos de interação, 287
 Medição, *veja também*
 Métricas, 539, 640, 586
 Medida, 539
 Melhora de processo (*veja* Melhora de Processo de Software)
 Mensagens, 747
 Metáforas, 308
 Método de Análise de dos Prós e Contras de uma Arquitetura (*Architecture Trade-Off Analysis Method - ATAM*), 245
 Método Dinâmico de Desenvolvimento de Sistemas (*Dynamic Systems Development Method - DSDM*), 84
 Metodologia de processo 40, 53
 Métodos formais, 69, 492
 conceitos, 500
 exemplos de, 500
 linguagens, 504
 notação matemática, 503
 Métricas,
 aperfeiçoamento do processo de software, 584
 argumentos favoráveis, 597
 atributos das, 542
 casos de uso, 592
 complexidade ciclomática, 434
 componentes, 554
 confiabilidade, 395
 definição de, 538
 estabelecendo um programa, 599
 etiqueta, 585
 interface de usuário, 557
 linha de base, 597
 manutenção, 562
 modelo de projeto, 547
 morfologia, 548
 orientada a classe, 549
 orientada a objeto, 549, 591
 orientada a tamanho, 588
 orientadas a função, 589
 pequenas organizações, 598
 processo, 584
 produto, 538
 projeto de arquitetura, 547
 projeto, 586
 públicas e privadas, 585
 qualidade de especificação, 546
 qualidade de software, 594
 reconciliando LOC e FP, 588
 requisitos, 543
 revisões técnicas, 376
 SQA, 392
 terminologia, 539
 teste, 560
 WebApp, 557, 592
 Métricas de software (*veja* Métricas)
 Métricas para código, 559
 Mitos, 45
 Modelagem, 41, 116
 Modelagem ágil, 99
 Modelagem baseada em cenário, 155
 Modelagem baseada em classes, 166
 Modelagem de dados, 163
 Modelagem de requisitos, abordagens, 154
 entrada, 198
 estratégias, 181

- padrões, 192
 - princípios, 117
 - saída, 199
 - WebApp, 197
 - Modelagem orientada a fluxos, 182
 - Modelo Cascata, 59
 - Modelo Classe-Responsabilidade-Colaborador (*Class Responsibility Collaborator - CRC*), 171, 455
 - Modelo de análise (*veja também*, Modelo de análise), 142
 - baseado em cenário, 155
 - baseado em classe, 166
 - elementos de, 155
 - orientado a fluxo, 182
 - Modelo de classe, consistência, 455
 - Modelo de interação, WebApps, 200
 - Modelo de maturidade, 685
 - Modelo de Maturidade de Capacidade (*Capability Maturity Model - CMM*), 685
 - Modelo de navegação, 202
 - Modelo de padrões, 55
 - Modelo de processo evolucionário, 62, 68
 - Modelo de processo, ágil, 93
 - atividades, 53
 - concorrente, 67
 - especializado, 69
 - evolucionário, 62
 - fluxo de trabalho, 59
 - genérico, 71
 - incremental, 61
 - orientado a aspecto, 70
 - prescritivo, 58
 - Modelo de projeto, 221
 - dimensões do, 221
 - métricas, 547
 - Modelo de requisitos, 142
 - baseado em cenário, 155
 - baseado em classe, 166
 - elementos do, 142
 - levantamento, 143
 - métricas, 543
 - Modelo de usuário, 292
 - Modelo Espiral, 65, 293
 - Modelo funcional, WebApps, 200
 - Modelo mental, 292
 - Modelo V, 60
 - Modelos de comportamento, 188
 - teste de, 465
 - Modelos dinâmicos, 213
 - Modelo-Visão-Controlador (*Model-View-Controller - MVC*), 349
 - Modos de navegação (*Ways of navigating - WoN*), 350
 - Modularidade, 111, 214
 - Módulos críticos, 414
 - Mudança, 515
 - Multiplicidade, 176, 729
- N**
- Negligência, 368
 - Negociação, 128, 145
 - Notação de grafo de fluxo, 432
- O**
- Object constraint language (OCL)*, 505
 - exemplo de, 506
 - notação, 505, 741
 - Objeto de conteúdo, 199, 345
 - Objetos dados, 163
 - relacionamentos, 164
 - Objetos de configuração, 518
 - identificação, 522
 - OMG/CORBA, 281
 - OOHDM, 352
 - Operações, 170, 746
 - Ordem de Alteração de Engenharia (*Engineering Change Order - ECO*), 524
 - Orientada a objeto, análise, 154, 182
 - arquitetura, 237
 - modelos, 455
 - projeto, 218
- P**
- Pacotes, 178
 - Padrão de fases, 56
 - Padrão de projeto, 317
 - descrição do, 317, 320
 - granularidade, 317
 - objetivos, 317
 - template, 321
 - tipos de, 319
 - Padrões, 112
 - análise, 145, 192
 - arquitetura, 238, 32
 - criacionais, 320
 - descrição de, 320
 - generativos, 318
 - interface de usuário, 302, 331
 - modelagem de requisitos, 192
 - nível de componente, 329
 - projeto, 300, 316
 - repositório, 321
 - teste, 449
 - WebApps, 331
 - Padrões, baseados em componente, 262
 - Padrões de análise, 145, 192
 - Padrões de arquitetura, 238
 - Particionamento de equivalência, 441, 480
 - Particularidades, 115
 - Percepção de sistema, 291
 - Pessoal, 567, 722
 - esforços das, 632
 - problemas de comunicação, 573
 - Planejamento, 40
 - distribuição de esforços, 634
 - princípios, 114
 - XP, 88
 - Planejamento de incremento, 492
 - Planejamento de projeto, 604
 - conjunto de tarefas, 606
 - estimativa, 605
 - recursos, 607
 - Plano de contingência, 657
 - Plano RMMM, 658
 - Polimorfismo, 474
 - Ponto de Função (*Function Point - FP*), 543, 590
 - estimativa da, 614
 - Pontos âncora de controle, 65
 - Pontos de objeto, 619
 - Pontos de prioridade, 132
 - Portabilidade, 363
 - Pós condição, 501
 - Práticas, 42

- descrição das, 109
 - essência das, 42
 - princípios, 11
 - Pré-condição, 501
 - Preocupações, 217
 - Princípio Comum da
 - Reutilização (*Common Reuse Principle - CRP*), 265
 - Princípio da Inversão da Dependência (*Dependency Inversion Principle - DIP*), 263
 - Princípio de Equivalência de Reúso de Versões (*Release Reuse Equivalency Principle - REP*), 265
 - Princípio de Fechamento Comum (*Common Closure Principle - CCP*), 265
 - Princípio de Segregação de Interfaces (*Interface Segregation Principle - ISP*), 264
 - Princípio de Substituição de Liskov (*Liskov Substitution Principle - LSP*), 263
 - Princípio do Aberto-Fechado (*Open-Close Principle - OCP*), 263
 - Princípio W5HH, 578
 - Princípios de codificação, 120
 - Processo,
 - adaptação, 41
 - ágil, 84
 - decomposição, 576
 - dualidade do, 77
 - elementos do, 40
 - integração de métricas, 596
 - nível comercial, 665
 - padrões, 55, 58
 - princípios, 110
 - problemas de gerenciamento, 575
 - tendências, 710
 - Processo de projeto, 209
 - Processo de software (*veja* Processo)
 - Processo de Software em Equipe (*Team Software Process - TSP*), 75, 695
 - Processo de Software Pessoal (*Personal Software Process - PSP*), 74, 696
 - Processo Unificado, 71
 - fases do, 72
 - versão ágil, 101
 - Produto, 567, 574
 - Produtor, 381
 - Programação em dupla, 90, 380
 - Programação estruturada, 275
 - Programação Extrema (*Extreme Programming - XP*), 87
 - codificação, 90
 - debate sobre, 92
 - elementos-chave, 87
 - industrial XP, 91
 - lanejamento, 87
 - processo, 88
 - projeto, 89
 - teste, 90
 - Projeto, 206, 568, 577
 - baseado em padrão, 214, 316, 322
 - centrado no usuário, 291
 - conceitos, 212
 - conjunto de tarefas genéricas, 222
 - da arquitetura, 239
 - descrição 230
 - diretrizes, 265
 - em nível de componente, 224, 257
 - etapas de projeto, 269
 - evolução do, 221
 - granularidade, 334
 - interface de usuário, 287, 292, 300, 311
 - nível de implantação, 224
 - notação gráfica, 276
 - notação tabular, 377
 - orientado a objeto, 218
 - orientado por domínio, 91
 - pós moderno, 714
 - princípios, 118
 - princípios, 262
 - visando a reutilização, 282
 - WebApp, 274, 338, 341
 - XP, 88
 - Projeto da interface abstrata, 353
 - Projeto da navegação, WebApps, 350
 - Projeto de conteúdo, 274, 345
 - Projeto de dados, 222
 - Projeto de software, *veja também* Projeto, 206
 - Projeto em nível de componente (*veja* Projeto, em nível de componente)
 - Projeto estético, 344
 - Projeto estruturado, 247
 - Projeto formal, 492
 - Projeto funcional, 275
 - Projeto orientado por domínio, 91
 - Projeto Visando a Reutilização (*Design For Reuse - DFR*), 282
 - Prototipação, 62, 319
 - Prova de exatidão, 497
 - Pseudocontroladores (*Drivers*), 408
 - Pseudocontrolados, 409
- Q**
- Qualidade, *veja também*
 - Qualidade de software, diretrizes, 209
 - gerenciamento, 357, 389
 - medição, 594
 - métricas, 594
 - visões da, 359
 - Qualidade de software, 210
 - avaliando, 363
 - custo da, 366
 - custos relativos, 367
 - definição da, 360
 - dilema, 365
 - dimensões da, 360
 - diretrizes de design, 209
 - elementos da, 360
 - fatores de McCall, 361
 - impacto das ações de gerenciamento, 369
 - obtendo, 370
 - visão quantitativa, 364
 - Qualidade do projeto, avaliação da, 210
 - Questões, livres de contexto, 132
- R**
- Recuperação do projeto, 670
 - Recursos, 607
 - Recursos de ajuda, 303

- Recursos via Internet (*Net sourcing*), 35
 - Reengenharia, *veja também*
 - Reengenharia de software, 496
 - Reengenharia de Processo de Negócio (*Business Process Reengineering - BPR*), 665
 - Reestruturação de documento, 669
 - Reestruturação do código, 670
 - Reestruturação dos dados, 670
 - Reestruturando, 674
 - Refatoração, 89, 218, 270
 - Referenciais,
 - banco de dados de projeto, 518
 - definição de, 517
 - Refinamento, 217
 - Regras de ouro, 288
 - Relatório de status, 527
 - Relatórios, 534
 - Repositório, 283, 519
 - características, 519
 - hipermídia, 335
 - padrões de design, 327
 - papel do, 519
 - Repositório de software (*veja* Repositório)
 - Representações de estado, 189
 - Requisitos,
 - acompanhando, 520
 - emergente, 816
 - negociando, 145
 - validando, 1460
 - Responsabilidade Civil, 368
 - Responsibilidades, 171
 - diretrizes de alocação, 171
 - Retirado (*Checkout*), 524
 - Reunião de revisão, 381
 - Reunindo requisitos, 132, 492
 - Reutilização, 282
 - Revisão da configuração, 417
 - Revisões, 210, 373
 - analisando a eficiência, 377
 - controladas por amostragem, 345
 - dados de inspeção, 378
 - diretrizes, 343
 - eficácia dos custos, 377
 - espectro de formalidade, 379
 - informais, 340
 - lista de problemas, 343
 - lista de verificação, 341
 - manutenção de registros, 343
 - métricas para, 376
 - modelo de referência, 379
 - relatório resumido, 343
 - reportando, 343
 - trabalho de desenvolvimento, 378
 - walkthroughs, 342
 - Revisões técnicas (*veja* Revisões)
 - Revisões técnicas formais, *veja também* Revisões, 381
 - Riscos, 368
 - avaliando, 650
 - componentes e fatores, 650
 - exposição aos, 655
 - formato CTC, 657
 - identificação dos, 649
 - impacto dos, 655
 - moderação, 657
 - projeção, 651
 - refinamento, 657
 - tipos de, 649
- S**
- SCAMPI, 58
 - Scrum*, 95
 - Segurança, 339, 368, 594
 - Segurança do software, 396, 658
 - Seis Sigma, 392
 - Semântica de navegação, 350
 - Separação por interesse, 110, 214
 - Sintaxe de navegação, 351
 - Sistema de forças, 317
 - Software “bom o suficiente”, 356
 - Software aberto, 35, 709
 - Software legado, 36
 - Software,
 - aberto, 703
 - blocos básicos, 708
 - características, 32
 - código aberto, 709
 - definição do, 32
 - domínios de aplicação, 34
 - importância do, 721
 - mitos, 45
 - morte do, 30
 - natureza do, 31
 - questões sobre, 32
 - visão do economista, 52
 - Solicitação de alteração, 524
 - Solução Spike, 89
 - SPI Estatístico, 586
 - SPICE, 58, 696
 - Subclasse, 746
 - Superclasse, 746
 - Supportabilidade, 664
- T**
- Tabela de ativação de processo, 187
 - Tabela de organização de padrões, 325
 - Tabela de riscos, 653
 - Tabelas de decisão, 277
 - Tarefa, 40
 - análise, 295
 - elaboração, 296
 - padrão, 55
 - rede, 636
 - Tecnologia,
 - ciclo da excelência, 704
 - ciclo de inovação, 703
 - evolução, 702
 - administrando a complexidade, 705
 - grande desafio, 711
 - tendências, 701, 709
 - visão no longo prazo, 724
 - Tecnologia de processo, 76
 - Tempo de colocação no mercado (*Time-to-market*), 340
 - Tempo de resposta, 303
 - Tempo Médio Entre Falhas (*Mean-time-between-failure - MTBF*), 396
 - Tempo Médio Para Reparar (*Mean-time-to-repair - MTTR*), 396
 - Tendências, 704
 - Terceirização, 625
 - Testabilidade, 429
 - Testando,

- abordagem estratégica, 402, 406
 - base de dados, 473
 - baseado em estado, 464
 - baseado em gráfico, 440
 - caixa branca, 431
 - caixa preta, 440
 - caminhos de navegação, 481
 - ciclos, 438
 - cliente/servidor, 446
 - com base em cenário, 460
 - com base em falhas, 459
 - com base em modelos, 445
 - com base em sequências de execução, 415, 458
 - com base no uso, 415, 458
 - conteúdo, 471
 - critério de finalização, 405
 - de forma especializada, 446
 - desempenho, 485
 - documentação, 447
 - estatisticamente, 493, 498
 - estrutura de controle, 437
 - estrutura profunda, 462
 - estrutura superficial, 462
 - exaustivamente, 431
 - fluxo de dados, 438
 - fundamentos, 429
 - GUIs, 446, 475
 - hierarquia de classe, 460
 - métricas, 560
 - modelos de comportamento, 465
 - múltiplas classes, 463
 - nível de componente, 480
 - organização, 403
 - padrões, 449
 - princípios, 120
 - recursos de ajuda, 447
 - segurança, 484
 - sistemas em tempo real, 448
 - software convencional, 407, 428
 - software orientado a objeto, 415, 453, 457
 - WebApp, 416, 468, 483
 - XP, 90
 - Teste aleatório, 462
 - Teste alfa, 417
 - Teste baseado em uso, 415, 458
 - Teste beta, 417
 - Teste caixa branca, 431
 - Teste caixa preta, 439
 - Teste da caixa de vidro, 431
 - Teste de agregado, 416, 458
 - Teste de base de dados, 473
 - Teste de caminho, 480
 - Teste de caminho básico, 431
 - Teste de carga, 486
 - Teste de ciclo, 438
 - Teste de classe, 457
 - Teste de condição, 437
 - Teste de configuração, 483
 - Teste de conteúdo, 471
 - Teste de estrutura de controle, 437
 - Teste de fluxo de dados, 438
 - Teste de fumaça, 413
 - Teste de integração, 409
 - ascendente, 411
 - contexto orientado a objeto, 415, 457
 - descendente, 410
 - primeiro-em-largura, 410
 - primeiro-em-profundidade, 410
 - produtos acabados, 414
 - Teste de navegação, 481
 - Teste de partição, 463
 - Teste de recuperação, 419
 - Teste de regressão, 412
 - Teste de segurança, 419, 483
 - Teste de sensibilidade, 419
 - Teste de sistema, 418
 - Teste de software (veja Teste)
 - Teste de uso estatístico, 498
 - Teste de validação, 417
 - contexto orientado a objeto, 458
 - critério de teste, 417
 - Teste do matriz ortogonal, 422
 - Teste por esforço, 419, 485
 - Teste por esforço, 419, 486
 - Teste unitário, 407
 - considerações, 407
 - contexto orientado a objeto, 415, 457
 - Testes de compatibilidade, 478
 - TickIT, 696
- U**
- UML, 727
 - diagrama de atividade, 161, 202, 272, 737
 - diagrama de caso de uso, 731
 - diagrama de componente, 224
 - diagrama de estado, 273, 740
 - diagrama de raiais (swimlane), 162, 298, 738
 - diagrama de sequência, 191, 734
 - estereotipo, 177, 178, 728
 - generalização, 728
 - história da, 727
 - notação do diagrama de classe, 728
 - notação do diagrama de comunicação, 735
 - notação do diagrama de implantação, 225, 731
 - OCL, 741
 - realização, 728
 - Unidades Semânticas da Navegação (*Navigation semantic units - NSUs*), 350, 482
 - Uso, 33
 - Usuários, tipos de, 292
 - Usuários finais, 113
 - Utilização, 287, 291, 363
 - projeto, 223
 - questões, 478
 - testes, 477
- V**
- Validação, 120, 129, 402
 - Velocidade de projeto, 88
 - Verificação, 402
 - Verificação de projeto, 497
 - Versão, 520
 - Vital, poucos, 393
- W**
- Walkthroughs* (veja Revisões)
 - WebApps, 529
- X**
- XP (veja Extreme Programming)